Objektorientierte Sichten zur Integration statisch getypter Objektbanken

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
vorgelegt von Diplom-Informatiker

Ralph Busse

geboren in Bad Harzburg

Referent: Prof. Dr. Erich J. Neuhold
Korreferent: Prof. Dr. Karl Aberer

Tag der Einreichung: 22. Dezember 2003
Tag der mündlichen Prüfung: 21. April 2004

Darmstädter Dissertation
D17
Darmstadt 2004

Zusammenfassung

Durch ihre mächtigen Fähigkeiten zur semantischen Modellierung haben objektorientierte und objektrelationale Datenbanksysteme in den letzten Jahren auch kommerziell an Bedeutung gewonnen. Sie erlauben es, einen Sachverhalt anschaulich und relativ „natürlich“ in der Datenbank darzustellen. Bei den meisten Systemen ist diese Modellierungsfähigkeit allerdings auf die Designphase beschränkt. Sobald die Datenbank vollständig entworfen ist, sind Veränderungen oft nicht, oder nur sehr umständlich möglich. Diese Veränderungen sind aber häufig notwendig, um die Datenbank an unterschiedliche Anwendungen anzupassen, oder um Weiterentwicklungen im Laufe der Zeit Rechnung zu tragen. Im Bereich relationaler Datenbanken stellt das Konzept der *Sicht* bzw. des *externen Schemas* eine zusätzliche Ebene für solche Anpassungen zur Verfügung. In der objektorientierten Welt ist dieses Konzept jedoch noch nicht vollständig entwickelt und wird, wenn überhaupt, nur mit Minimalfähigkeiten angeboten. Es gibt zwar einige Untersuchungen zur Natur und Funktionalität objektorientierter Sichten, aber diese Ansätze werden bisher noch in keinen Produkten unterstützt. Vielmehr treffen viele dieser Arbeiten Annahmen über die zugrunde liegenden Systeme, die von vielen kommerziellen Systemen überhaupt nicht erbracht werden können.

Diese Arbeit untersucht und klassifiziert die verschiedenen Ansätze und setzt sie mit den Fähigkeiten in Beziehung, die von existierenden Datenbankimplementierungen angeboten werden können. Dabei wird speziell auf den objektorientierten Datenbankstandard ODMG eingegangen, der wie C++ den Einschränkungen einer kompilierenden, statisch getypten Programmiersprache unterliegt. Ausgehend von dieser Analyse wird ein Sichtkonzept entwickelt, das maximale Funktionalität zur Verteilung und Integration von Objektbanken anbietet und dennoch auf einem standardkonformen C++-System implementiert werden kann. Eine solche Implementierung ist im Rahmen des IRO–DB-Projekts durchgeführt worden und wird ebenfalls in dieser Arbeit dokumentiert. Durch die geringen Anforderungen an das zugrunde liegende System kann dieser Ansatz auch auf andere Systeme übertragen werden, was exemplarisch anhand von CORBA gezeigt wird.

Hinweis: Aus technischen Gründen folgt diese Arbeit noch den Regeln der alten Rechtschreibung.

Wissenschaftlicher Werdegang des Verfassers in Kurzfassung

1986–1991	Studium der Informatik an der Technischen Universität Clausthal
5. Sept. 1991	Abschluß als Diplom-Informatiker
1991–2002	Wissenschaftlicher Mitarbeiter am Institut für Integrierte Publikations- und Informationssysteme (IPSI) in Darmstadt

Abstract

In the recent years, object-oriented and object-relational database systems gained commercial significance. This is due to their powerful capabilities in semantic modeling, allowing for descriptive and rather “natural” representations of real life facts. In most of the systems, however, this modeling capability is restricted to the design phase. As soon as the database is completely set up, design changes are very cumbersome, if not impossible. Such changes are, however, necessary in order to adapt the databases to different applications or to keep pace with ongoing development. In the field of relational databases, an additional adaptive layer is realized in the form of the *external schema* or *view* concept. In the object-oriented area, however, such a concept has not yet been established. If it is provided at all, it offers only minimum capabilities. In literature, there is some work on the nature and functionality of object-oriented views. None of these approaches have found their way into commercial products. In fact, many approaches have very high demands on the underlying systems, which cannot be accomplished by any existing commercial system.

This work examines and classifies the different approaches for object-oriented views and compares them with the abilities offered by existing database implementations. It dwells especially on the object-oriented database standard ODMG, which underlies the restrictions of a compiled and statically typed environment, just like the programming language C++. On the basis of this analysis a view concept is developed, which offers maximum functionality for distribution and integration of object bases and can nevertheless be realized on a standard-compliant C++ system. An implementation was accomplished in the context of the IRO–DB project and is documented in this work. Due to the little demands on the underlying system, this approach can be transferred to other systems as well, which is shown exemplarily for CORBA.

Danksagung

Diese Arbeit entstand am ehemaligen GMD-Institut und jetzigen Fraunhofer Institut für Integrierte Publikations- und Informationssysteme (IPSI) in Darmstadt während der Arbeit an mehreren Kooperationen und EU-Projekten.

Ich danke Herrn Professor Dr. Erich J. Neuhold für die Möglichkeit, die Dissertation an diesem Institut durchführen zu dürfen, sowie für die wertvollen Hinweise zur Verbesserung der vorliegenden Arbeit. Herrn Professor Dr. Karl Aberer danke ich für die Bereitschaft, als Korreferent zur Verfügung zu stehen, obwohl er bereits seit längerer Zeit an der EPFL in Lausanne unterrichtet und für meine Promotion nach Darmstadt anreisen mußte. Außerdem danke ich ihm für die jahrelange, kontinuierliche Motivation zur Fertigstellung der Arbeit. Weiterer Dank gilt den Mitgliedern der Prüfungskommission, Herrn Professor Dr. Oskar von Stryk, Herrn Professor Dr. Thomas Kühne und Herrn Dr. Matthias Müller-Hannemann für die Einarbeitung in meine Dissertation und die Möglichkeit zu vorbereitenden Gesprächen.

Mein besonderer Dank gilt Herrn Dr. Peter Fankhauser für die Betreuung meiner Arbeit über die gesamte Entwicklungszeit hinweg. Es waren in erster Linie die Diskussionen mit ihm und seine Vorschläge und Anmerkungen, die mich durch alle Phasen meiner Dissertation begleitet haben und zu einer fundierten und ausgereiften Arbeit geführt haben. Weiterhin danke ich meinen Kollegen vom Bereich Informationsmanagementsysteme (OASYS) am IPSI für viele fruchtbare Diskussionen und die gute Zusammenarbeit. Insbesondere danke ich meinem langjährigen Kollegen Herrn Gerald Huck, mit dem ich viele lange, schöpferische Diskussionen geführt, viele Projekte durchgeführt und unlösbare Probleme gelöst habe.

Von hohem Wert waren auch die Diskussionen und Entwicklungsarbeiten in verschiedenen EU-Projekten; allen voran natürlich IRO-DB, wo die Wurzeln dieser Arbeit gelegt wurden. Hier gilt mein Dank allen beteiligten Mitarbeitern von Ibermática/Spanien, EDS/Frankreich, Euriware/Frankreich, Intrasoft/Griechenland, Gopas/Deutschland, der Universität Versailles und dem FAW Linz. Viele Probleme wurden in gemeinsamen Sitzungen gelöst und die Realisierung des gesamten Systems basiert auf der guten Zusammenarbeit aller Partner.

Zum Schluß möchte ich meinen Eltern danken, ohne deren Unterstützung mein Studium und die erfolgreiche Promotion nicht möglich gewesen wären, sowie meinem jetzigen Arbeitgeber, Herrn Dr. Marcus Rumpf, für die Möglichkeit, die Fertigstellung der Dissertation mit der Arbeit in Einklang zu bringen.

Ralph Busse

Darmstadt, April 2004

Inhaltsverzeichnis

Definitionsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Ziel dieser Arbeit ist die Entwicklung eines Konzepts zur Definition von sogenannten *Sichten* für objektorientierte Datenbanken. Hierbei werden neben den allgemeinen Problemen objektorientierter Sichten auch die speziellen Forderungen bei deren Einsatz zur Integration und Interoperabilität verteilter Datenbanken betrachtet. Neben diesen theoretischen Aspekten ist die Implementierbarkeit des Konzepts ein ausdrückliches Ziel der Arbeit. Sie orientiert sich deshalb am Datenbank-Referenzmodell ODMG–93 und dessen Sprachbindung für die objektorientierte Programmiersprache C++. Im folgenden wird eine Beschreibung der aktuellen Situation gegeben, die vorliegende Arbeit motiviert und der Beitrag dieser Arbeit zusammengefaßt.

1.1 Hintergrund

Objektorientierte Datenbanken

In heutiger Zeit werden zur Datenhaltung vorwiegend *relationale Datenbanken* eingesetzt. Basierend auf einer gut ausgearbeiteten Theorie von E. F. Codd ([Cod70]) stellen sie die gespeicherten Daten in Form von Tabellen zur Verfügung, wobei jeder Datensatz aus einem Tupel flacher Werte besteht. Durch die starke theoretische Grundlage besitzen die relationalen Datenbanken eine formale Semantik, die es erlaubt, Strukturen, Anfragen und Optimierungen formal zu beschreiben und ihre Eigenschaften zu beweisen. Durch die dadurch möglichen Optimierungen bilden die relationalen Datenbanken eine sehr effiziente Grundlage zur Verarbeitung großer Mengen einzelner Datensätze. Ihr prinzipieller Nachteil ist die Beschränktheit auf flache, statische Daten. Die Daten enthalten keine weitergehende Funktionalität, kein Wissen über sich selbst; ihre Interpretation liegt vollständig in der Hand der Anwendungsprogramme. Weiterhin ist ihre Struktur auf das Tupelkonzept beschränkt. Tiefere Strukturierungen der Daten sind entweder nicht möglich, oder müssen über einen Zwischenschritt auf die flache Tabellenstruktur abgebildet werden (NF^2-Modell, siehe z.B. [Vos87]).

Um dieser Einschränkung abzuhelfen, begann man Anfang der achtziger Jahre, Konzepte aus dem Bereich der objektorientierten Programmiersprachen auf Datenbanken anzuwenden ([ABC*83], [ZAB*84]). Die wesentlichen Erweiterungen zu den Standard-Datenbankeigenschaften bestehen aus der Kapselung von Daten und Operationen in Objekten und der Einführung einer Objektidentität, die von Änderungen an den Objektinhalten nicht beeinflußt wird. Im Gegensatz zu den relationalen Datenbanken fehlt bei diesen *objektorientierten Datenbanken* ein einheitliches theoretisches Modell. Die Systeme sind unabhängig voneinander aus relationalen Datenbanken oder aus objektorientierten Programmiersprachen entstanden. Abgesehen von den grundsätzlichen Konzepten gibt es nur geringe Gemeinsamkeiten in den verschiedenen Systemen. Erste Kriterienkataloge wurden ab 1989 erarbeitet ([ABW*89], [Bee89], [Zdo93]); der einzige Ansatz für einen objektorientierten Standard ist 1993 von der Object Database Management Group ODMG unter dem Namen ODMG–93 veröffentlicht worden und liegt inzwischen in der Version 3.0 vor ([Cat93-96], [CaB97-00]). Da die ODMG in erster Linie aus Vertretern der Datenbankindustrie besteht, hat der Standard auch eine kommerzielle Bedeutung und wird im allgemeinen als Referenzmodell für objektorientierte Datenbanken herangezogen. ODMG–93 wird in Kapitel 2.1 näher beschrieben. Die notwendigen Anpassungen für neuere Versionen werden in Kapitel 6 behandelt.

Sichten

Allen Datenbanken gemeinsam ist die Eigenschaft, daß die abgelegten Daten in einem bestimmten *Schema* vorliegen, das die Struktur der Daten beschreibt. Gemäß ANSI-Modell können solche Schemata in verschiedenen Ebenen auftreten, wobei die äußerste Ebene, das *externe Schema*, die Datenbank an die Anforderungen einzelner Benutzer oder Programme anpassen kann (siehe Architekturbild auf Seite 17). Diese externen Schemata werden auch *Sichten* genannt, da sie jeweils eine spezifische Sichtweise auf die gemeinsamen Daten darstellen. Im relationalen Modell sind diese Sichten standardisiert und weit verbreitet. Im objektorientierten Modell führt die Reichhaltigkeit an Modellierungsmöglichkeiten jedoch zu großen konzeptuellen Problemen bei der Umsetzung dieses Konzepts. Die bisher veröffentlichten Arbeiten zu objektorientierten Sichten beziehen sich fast immer nur auf theoretische Datenmodelle. In kommerziellen objektorientierten Datenbanken sind Sichten praktisch nicht vorhanden, und die wenigen vorhandenen Implementierungen sind meist auf spezielle Prototypen zugeschnitten und sehr eingeschränkt. Bei den erwähnten theoretischen Arbeiten ist zu beobachten, daß in den verwendeten Modellen häufig Annahmen getroffen werden, die von kommerziellen objektorientierten Datenbanken fast nie erfüllt werden.

Datenintegration und Interoperabilität

Ein spezielles Anwendungsgebiet von Sichten sind die *Interoperabilität* und die *Integration* von autonomen Datenbanken. Interoperabilität beschreibt allgemein die Fähigkeit, Daten zwischen zwei Datenbanken austauschen zu können. Hierbei können Sichten die Schemata der beiden Datenbanken aneinander anpassen, um den eigentlichen Datenaustausch zu vereinfachen. Zusätzlich kann auch der Netzwerkzugriff auf eine entfernte Datenbank in einer lokalen Sicht verborgen werden. Das Ziel der Integration ist hingegen, einen gemeinsamen, integrierten Zugriff auf mehrere Datenbanken zur Verfügung zu stellen. Die verschiedenen Schemata werden also an ein übergreifendes Schema angepaßt. Ein Hauptproblem bei der Datenbankintegration ist der Entwurf eines integrierten Schemas, das die Daten der beteiligten Datenbanken effizient und sinnvoll zusammenfügt. Zu diesem Thema gibt es schon seit Jahren umfangreiche Arbeiten, die die Probleme und Lösungsmöglichkeiten diskutieren (siehe zum Beispiel [BLN86], [ShL90], [PaS00]). Dieser Designprozeß selbst ist jedoch für die vorliegende Arbeit nur von geringer Bedeutung. Sofern es erforderlich ist, beziehe ich mich auf die am IPSI entwickelte Integrationsmethodologie von P. Fankhauser ([Fan97]), die auf Basis von Korrespondenzdefinitionen und mit Hilfe struktureller Anreicherungen eine verlustfreie, objektorientierte Integrationsphase zur Verfügung stellt. Neben diesem Designproblem muß die gefundene Abbildung aber auch zur Laufzeit angeboten werden, um dem Nutzer die Illusion einer einzelnen großen Datenbank zu liefern. Im Gegensatz zum sogenannten *Data Warehouse*, bei dem die Daten nach der Integration in eine separate Datenbank mit dem gewählten Schema kopiert werden, folgt diese Arbeit eher dem *Mediator*-Ansatz, bei dem transparent direkt auf die Originaldatenbanken zugegriffen wird.

Die Programmiersprache C++

Die Datenbankmodelle beschreiben in erster Linie nur ein Konzept zur Datenhaltung, und die eigentliche Implementierung ist darüber hinaus an eine konkrete Programmiersprache gebunden. Neben Java ist C++ die am weitesten verbreitete objektorientierte Programmiersprache. Ein Großteil der verfügbaren kommerziellen Datenbanken ist in C++ implementiert worden und kann auch meist nur in C++-Umgebungen verwendet werden. Durch die strenge Typisierung und den Übersetzungsschritt durch einen Compiler folgen jedoch teilweise erhebliche Einschränkungen in der verfügbaren Funktionalität.

1.2 Motivation

Wie gerade beschrieben zählen objektorientierte Datenbanken und die Programmiersprache C++ zu den wesentlichen Elementen heutiger Datenhaltung und Softwareentwicklung. Auch Interoperabilität und Integration sind gerade in heutiger Zeit von großer Bedeutung: Durch das große Marktangebot hat sich in den meisten Firmen bereits eine hohe Vielfalt an gleichzeitig verwendeten Systemen gebildet. Der Aufwand, diese unterschiedlichen Systeme nun im Zuge einer umfassenderen Produktionsunterstützung miteinander Daten austauschen zu lassen, ist bereits beträchtlich. Gerade der Umgang mit diesen Altlasten, den sogenannten *Legacy Systems*, hat mit zur Entwicklung der Data Warehouses beigetragen. Diese bieten jedoch nur reine Abfragefunktionalität. Man kann also die verschiedenen, unabhängig voneinander vorliegenden Daten zueinander in Beziehung setzen und mit statistischen Methoden auswerten. Es fehlt jedoch die Komponente des direkten, zeitgleichen Zugriffs, und es gibt keine Möglichkeit, die Daten aus der integrierten Kopie heraus zu verändern. Angesichts dieser Probleme überrascht es, daß die objektorientierten Datenbanken mit ihrer mächtigen Ausdruckskraft noch keine Sichtkonzepte unterstützen. Die hier beschriebene Arbeit versucht nun, diese Lücke zu füllen. Es wird der industrierelevante Datenbankstandard ODMG–93 verwendet, um auf C++ ein implementierbares Sichtsystem zu entwerfen, das den Datenaustausch und die Integration von objektorientierten Datenbanken unterstützt.

1.3 Eigener Beitrag

Eine Untersuchung der veröffentlichten Arbeiten zum Thema objektorientierter Sichten zeigt, daß die meisten konzeptuellen Arbeiten die Fähigkeiten der existierenden kommerziellen objektorientierten Datenbanken fast völlig außer Acht lassen. In den meisten Fällen ist entweder nur die Theorie selbst von Interesse, oder das Konzept ist speziell für die Fähigkeiten eines vorhandenen Forschungsprototyps entwickelt worden. Die Arbeit gibt eine Übersicht über die bisherigen Arbeiten und beschreibt die auftretenden Probleme. Basierend auf dem objektgenerierenden Sichtansatz von S. Heiler und S. Zdonik ([HeZ90]) wird ein *verallgemeinertes identitätserhaltendes Sichtkonzept* entwickelt, das es erlaubt, in einer objektgenerierenden Umgebung die Identität der Originalobjekte in die Sichtobjekte zu übernehmen. Es stellt somit eine wichtige Brücke zwischen Theorie und Praxis dar, indem es das objekterhaltende Konzept auf Basis verfügbarer objektgenerierender Systeme modelliert. Dieses Konzept wird auf den Datenbankstandard ODMG–93 angewandt und bietet neben allgemeiner Sichtfunktionalität auch Unterstützung für die Integration verteilter Datenbanken. Es werden verschiedene Möglichkeiten zur Spezifikation angeboten und die Probleme bei Updates und bei Methodenaufrufen untersucht. Um die Implementierbarkeit zu zeigen, wird eine komplette Arbeitsumgebung vorgestellt, die das Sichtkonzept für ein ODMG-System zur Verfügung stellt. Die Basisfunktionalität ist im Rahmen des ESPRIT-III-Projekts IRO-DB ([GGF*96], [GFF97]) implementiert worden. Die eigenen Beiträge lassen sich durch die folgenden Kernpunkte charakterisieren:

1. Anforderungsanalyse für objektorientierte Sichten

Während Sichten im relationalen Modell standardisiert und theoretisch gut fundiert sind, gibt es im objektorientierten Bereich auf Grund der großen Mächtigkeit und Vielfalt objektorientierter Datenmodelle keinen einheitlichen Sichtansatz. Als Vorbereitung für den Entwurf eines eigenen Systems untersuche ich daher in Kapitel 2 die unterschiedlichen Ansätze und Varianten, nach denen ein objektorientiertes Sichtsystem entworfen werden kann. Ein Teilaspekt sind dabei die Anforderungen beim Einsatz eines solchen Sichtsystems zur Integration objektorientierter Datenbanken.

2. Entwurf eines konkreten objektorientierten Sichtmodells

Aus der Anforderungsanalyse ergeben sich eine Reihe von Kriterien, die beim Entwurf eines objektorientierten Sichtmodells beachtet werden müssen. Zum einen ist es wünschenswert, qualitative Eigenschaften relationaler Sichten beizubehalten, wie etwa Abgeschlossenheit bezüglich des Datenmodells, Unterstützung von Updates, Kompositionalität und Wahrung der Datenbankautonomie. Darüber hinaus muß aber der erweiterten Mächtigkeit objektorientierter Datenmodelle Rechnung getragen werden, indem Methoden, Kapselung, Vererbung und Objektidentität geeignet umgesetzt werden. Gleichzeitig muß man darauf achten, daß die Anforderungen, die sich an eine Implementierung ergeben, noch erfüllbar bleiben. Mit Hilfe dieser Kriterien entwickle ich in Kapitel 3 ein eigenes Sichtsystem, das sich durch die folgenden Eigenschaften auszeichnet:

- Die Sicht ist unter dem Datenmodell *abgeschlossen*.

  Eine Sicht besteht genau wie die Originaldatenbank aus einer Menge von Objekten. Das Datenmodell wird also nicht verlassen. Im Gegensatz zum relationalen Modell ist diese Abgeschlossenheit wesentlich schwerer zu erreichen, da sie die volle Mächtigkeit des objektorientierten Modells beachten muß.

- Die Sicht bietet *volle Sprachmächtigkeit*.

  Methoden sind ein wesentlicher Aspekt des objektorientierten Modells. Durch die Definition von Sichten in Form neuer Klassen können den Sichtobjekten beliebige Methoden hinzugefügt werden. Bei einem rein anfragebasierten System wäre dies aufgrund der eingeschränkten Mächtigkeit der verfügbaren Anfragesprachen nicht möglich.

- Die Sicht ist in sich *abgeschlossen*.

  In dem vorgestellten Sichtsystem ist eine Sicht stets ein komplettes Schema. Alle Klassen einer Sicht werden in einen eigenen Verband eingeordnet, der vom Klassenverband der Originaldaten unabhängig ist. Dies hat folgende Vorteile: i) Es wird verhindert, daß ein Benutzer gleichzeitig auf dem Sichtobjekt und dem zugehörigen Originalobjekt arbeitet, was zu Inkonsistenzen führen könnte. ii) Referentielle Integrität wird gewährleistet: Verfolgt man eine Relationship, so gelangt man über die zugehörige Inverse wieder zum Startpunkt zurück. Bei einer Relationship zwischen einem Sichtobjekt S_1 und einem Originalobjekt O_2 könnte man zwar von S_1 nach O_2 navigieren, die zugehörige Inverse führte auf Grund der Definition von O_2 aber nicht zurück zu S_1, sondern zu dessen Originalobjekt O_1. iii) Seiteneffekte durch die Inklusionsabhängigkeit im Klassenverband werden vermieden: Bei Einordnung der Sichtklassen in den Originalverband würden in den gemeinsam verwendeten Superklassen sowohl Objekte aus der Sicht als auch aus der Originaldatenbank vorkommen.

  Im Gegensatz zur Abgeschlossenheit unter dem Datenmodell existiert diese Art der Abgeschlossenheit im relationalen Modell nicht, da es dort keine expliziten Referenzen gibt. Eine relationale Sicht ist im wesentlichen eine einzelne Tabelle, deren Beziehung zu anderen Tabellen ausschließlich mit Hilfe von Datenvergleichen hergestellt wird.

- Sichten sind *kompositional*, sie können auf anderen Sichten definiert werden.

  Jede Sicht wird von einem Basisschema abgeleitet. Hierbei ist es unerheblich, ob das Basisschema ein Originalschema der Datenbank oder selbst eine Sicht ist.

- Die Sicht ist *verallgemeinert identitätserhaltend*.

  Das vom theoretischen Standpunkt wünschenswerte *objekterhaltende* Sichtkonzept, bei dem existierende Objekte dynamisch verschiedene Schnittstellen anbieten können, ist in der Praxis oft nicht umsetzbar. Besonders statisch getypte (compilierte) Sprachen stellen sehr wenig Funktionalität zur Verfügung; so wird zum Beispiel die Zuordnung zwischen Objekt und Objekttyp zur Übersetzungszeit bzw. zum Instanziierungszeitpunkt festgelegt und ist danach nicht mehr änderbar. Weitere Einschränkungen objekterhaltender Sichten sind die mangelnde Ausdrucksmächtigkeit in Bezug auf Gruppierungen sowie beim Zugriff auf externe Objekte.

  Der *objektgenerierende* Ansatz umgeht diese Probleme, indem die Sicht mit neu erzeugten Objekten gefüllt wird, die aus den Originalobjekten berechnet werden — hierbei geht jedoch die Objektidentität verloren.

  Der von mir vorgestellte Ansatz ist daher *verallgemeinert identitätserhaltend*. Er verwendet die objektgenerierende Technologie, um Sichten mit strukturellen Änderungen überhaupt zu ermöglichen, definiert aber darüber hinaus eine Zuordnungsfunktion, um eine semantische Identität von Sicht- und Originalobjekten bereitzustellen. Durch diese explizite Zuordnung von Original- zu Sichtobjekten kennt jedes Sichtobjekt seine Originalinstanz und kann direkt auf deren Daten zugreifen. Es übernimmt quasi die Identität des Originalobjekts innerhalb der Sicht. Zusätzlich zur direkten 1-zu-1-Abbildung zwischen Sicht- und Originalobjekten können auch Sichtobjekte für eingebettete Daten oder ganze Objektgruppen erzeugt werden. Im ersten Fall besteht dann eine Zuordnung zwischen dem Sichtobjekt und einem Attribut eines Originalobjekts, im zweiten Fall zwischen dem Sichtobjekt und der durch die Herleitung definierten Äquivalenzklasse der beteiligten Originalobjekte.

- Die Sicht unterstützt *Updates*.

  Obwohl Sichtanwender nicht direkt auf den Originalobjekten arbeiten, können dennoch Updates zur Verfügung gestellt werden. Auf Grund der Identitätserhaltung kann jedes Sichtobjekt intern auf die zugehörigen Originalobjekte zugreifen und dort die notwendigen Änderungen vornehmen. Bei einfachen Herleitungen kann die Update-Umsetzung gegebenenfalls automatisch bestimmt werden, bei komplexeren Sichtdefinitionen werden die notwendigen Operationen explizit beim Sichtentwurf spezifiziert.

  Die Fähigkeit zu Updates ergibt sich hier aus der Kombination von Objektidentität und Ausdrucksmächtigkeit der Sprache. Im relationalen Modell wird sie durch Analyse von Schlüsselabhängigkeiten bestimmt und kann nicht immer zugesichert werden.

- Die Sicht bewahrt die *Datenbankautonomie*.

  Durch die Definition neuer Objekte und deren Einordnung in einen unabhängigen Klassenverband ist keinerlei Änderung an den Originalobjekten notwendig. Den existierenden Klassen werden weder Schnittstellen noch Datenfelder hinzugefügt. Die Identitätserhaltung wird vollständig vom Sichtsystem übernommen.

- Der Ansatz kombiniert *deklarative und prozedurale Spezifikationen*.

  Die wesentlichen Elemente jeder Sicht werden deklarativ spezifiziert. Dies führt zu Kompaktheit und guter Lesbarkeit wie im relationalen Modell. Sind kompliziertere Herleitungen notwendig, die deklarativ nicht oder nur schwer ausdrückbar sind, so können diese prozedural hinzugefügt werden. Dies geht über die Fähigkeiten relationaler Sichten hinaus.

Beim Entwurf des Sichtsystems wird Wert darauf gelegt, daß es nicht nur als lokales Sichtsystem einer einzelnen Datenbank verwendet werden kann, sondern auch Datenbankinteroperabilität und -integration unterstützt. Hierbei erweist sich die Erzeugung neuer Objekte als unerläßlich, da externe Objekte im allgemeinen nicht direkt zugreifbar sind, sondern über Stellvertreterobjekte (*proxies*) angesprochen werden müssen. Das verallgemeinerte identitätserhaltende Sichtmodell stellt genau solche Objekte zur Verfügung, kümmert sich um deren Identität und erlaubt durch die Gruppierung von Objekten eine echte Unifikation von Objekten aus verschiedenen, heterogenen Datenbanken. Die folgenden vier Punkte zeigen spezifische Vorteile des Sichtmodells bei der Datenbankintegration:

- Die Sicht unterstützt den Zugriff auf *externe Objekte*.

  Im Sichtmodell werden spezielle Importklassen definiert, die ausschließlich dazu dienen, auf Objekte in anderen Datenbanken zuzugreifen. Auf dieser Ebene wird eine teilweise 1-zu-1-Abbildung verwendet, bei der bereits Datenfelder und Objekte ausgeblendet werden können.
- Die Sicht unterstützt die *Gruppierung von Objekten*.

  Bei der Integration überlappender Daten müssen Objekte geeignet gruppiert werden. Bei der *Unifikation* werden Objekte aus verschiedenen Datenbanken, die dasselbe Realweltobjekt darstellen, zu einem Objekt zusammengefaßt: Das Sichtobjekt repräsentiert eine Äquivalenzklasse; es ist ein Stellvertreter für jedes einzelne Originalobjekt. Bei der *Aggregation* werden unterschiedliche Objekte, die sich gegenseitig ergänzen, zu einem Objekt kombiniert: Das Sichtobjekt repräsentiert ein Tupel von Originalobjekten.
- Die Sicht unterstützt *Objektidentität* für externe Objekte.

  Beim Import externer Objekte in die Sicht wird darauf geachtet, daß für jedes externe Objekt genau ein Sichtobjekt angelegt wird. Dieses muß in der spezifischsten Klasse der Vererbungshierarchie erzeugt werden, in der es vorkommen kann.
- Die Sicht unterstützt unterschiedliche *Interoperabilitätsgrade*.

  Der Grad einer Datenbankintegration ist durch das Sichtmodell nicht festgelegt. Vom einfachen externen Datenzugriff über homogenisierte Schnittstellen zum Datenaustausch bis zum vollständig integrierten Zugriff ist alles darstellbar. Durch die Kompositionalität der Sichten kann eine vollständige Integration auch auf homogenisierenden Zwischenschritten basieren, wodurch die Komplexität der Sichtdefinition verringert werden kann.

3. Exemplarische Umsetzung für ein konkretes System

Um die Anwendbarkeit des Sichtmodells zu zeigen, ist es für ein konkretes System umgesetzt worden. Aus der Vielzahl objektorientierter Datenmodelle wurde hierfür ODMG–93 ausgewählt, da dies der einzige derzeit existierende Standard ist.[1] Als Basis für die Implementierung dient C++, da es zu den wichtigsten objektorientierten Programmiersprachen zählt und auch von ODMG–93 am stärksten beachtet wird. Modell und Sprache sind beide statisch getypt und erlauben keine objekterhaltende Sichtimplementierung. Da C++ derzeit noch keine Metainformation zur Laufzeit anbietet, muß zusätzlich zur identitätserhaltenden Objektverwaltung eine Arbeitsumgebung für dynamischen Datenzugriff realisiert werden. Die in Kapitel 4 beschriebene Umsetzung zeigt die Anwendbarkeit des Modells und zeichnet sich durch folgende Eigenschaften aus:

1. SQL3 befaßt sich nur mit objektorientierten Erweiterungen relationaler Systeme, ist also nicht auf rein objektorientierte Systeme anwendbar. CORBA und DCOM/OLE sind zwar objektorientiert, arbeiten aber nur auf einzelnen, direkt adressierbaren Objekten und nicht auf persistenten Objektmengen.

- Die Umsetzung ist *standardorientiert*.

  Sie basiert auf den wichtigen Datenbank- und Sprachstandards ODMG–93 und C++. Die Definitionssprache von ODMG–93 wird nur minimal erweitert, um Sichten ausdrücken zu können, an C++ werden keine Forderungen gestellt. Sowohl Datenmodell als auch Spezifikation und Implementierung orientieren sich streng an diesen Standards.

- Eine *Objektverwaltung* realisiert Identität und referenzielle Integrität.

  Für die Sichtobjekte wird eine vollständige Objektverwaltung zur Verfügung gestellt. Sie kümmert sich um Aktivierung und Deaktivierung der Sichtobjekte, bewahrt die Zuordnung zwischen Sichtobjekten und Originalobjekten, kümmert sich um Eindeutigkeit und Gültigkeit der Objekte und stellt referenzielle Integrität bei Relationships sicher.

- Wrapper erlauben *dynamischen, typsicheren Datenzugriff* zur Laufzeit.

  Durch den Compilationsschritt bietet C++ zur Laufzeit praktisch keine Metainformation für Daten und Objekte an. Besonders einschneidend ist der Mangel an dynamischem Datenzugriff, beispielsweise beim Auslesen eines Attributs, dessen Name und Typ erst zur Laufzeit bekannt sind. Durch die Erzeugung geeigneter Wrapperfunktionen wird eine solche Funktionalität während des Compilationsschritts hinzugefügt.

- Ein *Präprozessor* erzeugt automatisch schemaspezifischen Code.

  Gemäß ODMG–93 muß jede deklarative Schemadefinition in die entsprechende Sprachbindung überführt werden. Hierfür ist von mir ein Präprozessor entworfen worden, der neben der Original-Sprachbindung auch die notwendigen Methodenimplementierungen für Sichtklassen erzeugt.

1.4 Struktur der Arbeit

Die Arbeit ist wie folgt strukturiert: Das Grundlagenkapitel 2 beschreibt das verwendete Referenzmodell ODMG–93 und erläutert den Begriff der Sicht für relationale und objektorientierte Datenbanken sowie dessen Anwendung zur Interoperabilität und Integration von Datenbanken. Ausgehend von den grundsätzlichen Eigenschaften objektorientierter Datenmodelle wird ein Kriterienkatalog erarbeitet, der es ermöglicht, unterschiedliche objektorientierte Sichtansätze zu klassifizieren. Die folgenden beiden Kapitel bilden den eigentlichen Kern dieser Arbeit. In Kapitel 3 werden zuerst die Anforderungen formuliert, die an eine Sichtimplementierung gestellt werden müssen. Die rein sichtspezifischen Kriterien müssen dabei einigen Randbedingungen genügen, die sich aus der Mächtigkeit des objektorientierten Modells, der Unterstützung verteilter Datenbanken und der Realisierbarkeit ergeben. Ausgehend von diesen Kriterien wird ein Sichtmodell entwickelt, das diesen Anforderungen genügt und auf den Datenbankstandard ODMG–93 anwendbar ist. Die verwendete Spezifikationssprache kommt dabei mit minimalen Erweiterungen des Standards aus. In Kapitel 4 folgt eine detaillierte Beschreibung der Implementierung dieses Sichtkonzepts auf Basis von C++. Neben den allgemeinen Anforderungen aus Kapitel 3 muß man zusätzlich auf die Einschränkungen der Programmiersprache C++ eingehen sowie auf die verschiedenen Möglichkeiten, das Datenmodell der Datenbank auf das Datenmodell von C++ abzubilden. Kapitel 5 untersucht das beschriebene System in Bezug auf konkurrierende Zugriffe und die Ausführung externer Methoden. In Kapitel 6 wird der Ansatz auf andere Systeme übertragen. Neben der Untersuchung neuerer ODMG-Versionen wird auf andere Datenbanktypen eingegangen und es wird eine konkrete Übertragung des Systems auf die verteilte Objektumgebung CORBA vorgestellt. Ein Vergleich mit anderen Arbeiten im Bereich objektorientierter Sichten erfolgt in Kapitel 7. Kapitel 8 schließlich faßt die Arbeit zusammen und zeigt

mögliche Weiterentwicklungen. Die Anhänge im Anschluß an das Literaturverzeichnis enthalten Verweise auf Internetquellen, ein Glossar, Grammatiken und im Text verwendete Beispielschemata, sowie kommentierte Programmlistings.

2 Grundlagen: Objektmodell und Sichten

In diesem Kapitel beschreibe ich die notwendigen Grundlagen für meine Arbeit. Im ersten Teil wird das verwendete Objektmodell ODMG–93 beschrieben, das den Ausgangspunkt für das Datenmodell und alle Spezifikationen bildet. Anschließend erläutere ich den existierenden Sichtbegriff und seine Anwendungen und klassifiziere grundsätzliche Eigenschaften, die für das Verständnis objektorientierter Sichten notwendig sind. Auf konkrete Sichtmodelle wird im Zusammenhang mit verwandten Arbeiten in Kapitel 7 eingegangen.

2.1 Das Referenzmodell ODMG–93

Während relationale Datenbanken auf einer theoretischen Basis beruhen, existiert für objektorientierte Datenbanken kein einheitliches Datenmodell ([Zdo93]). Objektorientierte Datenbanken kombinieren Datenbankeigenschaften, wie Persistenz, Schemata und Transaktionen, mit Elementen objektorientierter Programmierung, wie Prozeduren, Kapselung und Objektidentität. Hieraus ergeben sich zwei grundverschiedene Wege zur Entwicklung objektorientierter Datenbanken: die strukturelle Erweiterung relationaler Datenbanken und das Hinzufügen von Persistenz zu objektorientierten Programmiersprachen. Dies führt zu einer Vielzahl völlig unterschiedlicher Systeme mit unterschiedlichen Datenmodellen, die jedoch alle mehr oder weniger auf denselben Konzepten beruhen. Diese Basiskonzepte objektorientierter Datenbanken wurden erstmals 1989 von Atkinson et al. im „Object-Oriented Database System Manifesto“ ([ABW*89]) zur Diskussion gestellt. Zu den wesentlichen Eigenschaften zählen komplexe Objekte mit Objektidentität und Kapselung, Typen und Klassen in einer Vererbungshierarchie, volle Programmiersprachenmächtigkeit mit Überladen von Methoden und dynamischem Binden, Erweiterbarkeit, Persistenz, Sekundärspeicherunterstützung, Parallelität, Fehlerbehebung sowie eine Anfragesprache. Optional können auch Mehrfachvererbung, Typinferenz, Verteilung, lange Transaktionen und Versionierung unterstützt werden. Ein formalerer Ansatz zur Definition objektorientierter Datenbankmodelle wird von Beeri in [Bee89] und [Bee90] gegeben. Auch hier finden sich die Begriffe Objekt, Klasse, Vererbung, Schema, Operation usw. wieder.

Trotz der Entwicklungen im objektorientierten Bereich gab es lange Zeit keinerlei Ansätze für einen objektorientierten Datenbankstandard, obwohl eine Standardisierung eine wesentliche Voraussetzung für den praktikablen Umgang mit objektorientierten Datenbanken ist: Allein durch die Mächtigkeit der objektorientierten Datenmodelle und ihrer engen Verknüpfung mit Programmiersprachen ist die Vielfalt der möglichen Implementierungen zu groß, um Portabilität und damit durchgängige Akzeptanz zu erreichen ([Cat94, S. 1]). Erst 1991 wurde die *Object Database Management Group (ODMG)* zum Entwurf eines Standards ins Leben gerufen. Erklärtes Ziel war dabei die Entwicklung produktunabhängiger Spezifikationen und Schnittstellen, um auf diese Weise portable Anwendungen zu ermöglichen. Bereits zwei Jahre später wurde die erste Version des *Object Database Standards ODMG–93* ([Cat93]) veröffentlicht. Nach diversen revidierten Fassungen ([Cat94], [Cat96], [CaB97]) ist die derzeit letzte Version 2000 unter dem Namen „The Object Data Standard: ODMG 3.0“ publiziert worden ([CaB00]). Dieser Standard, der sowohl ein generisches Objektmodell als auch Konformitätskriterien für die Umsetzung dieses Modells in konkrete Systeme definiert, wird allgemein als Referenzmodell herangezogen und bildet somit den einzigen de facto Standard für objektorientierte Datenbanken. Er dient daher auch als Basis für die vorliegende Arbeit.

2.1.1 Das ODMG-Datenmodell

Der ODMG–93-Standard besteht im wesentlichen aus zwei Teilen. Der erste Teil beschreibt ein generisches Datenmodell, bestehend aus der Spezifikation der verwendeten Daten- und Objekttypen, sowie der zu verwendenden Definitions-, Manipulations- und Anfragesprachen. Der zweite Teil enthält konkrete Implementierungsrichtlinien für Smalltalk und C++. Die nachfolgende Beschreibung basiert auf der Version 1.1, die zu Beginn der Arbeit vorlag. Die Auswirkungen der neueren Versionen werden in Kapitel 6.2 separat behandelt.

Das Datenmodell der ODMG ([Cat94], [LAC*93]) basiert auf dem bereits standardisierten Datenmodell aus der *Object Management Architecture* ([OMG95a]) und der *Common Object Request Broker Architecture (CORBA)* ([OMG91]) der *Object Management Group (OMG)*. Das CORBA-Modell wurde für verteilte Programmierumgebungen entworfen und konzentriert sich in erster Linie auf den Zugriff auf einzelne Objekte. Für den Einsatz im Datenbankbereich fehlen eine ausreichende mengenorientierte Auswertung sowie ein geeigneter Persistenzbegriff. Darüber hinaus erweitert ODMG–93 das Typsystem um weitere atomare und parametrisierte Typen und fügt Klassen zur Repräsentation von Datenbanken und Transaktionen hinzu. Die folgenden Definitionen geben einen Überblick über das Datenmodell. Hierbei werden teilweise die englischen Bezeichnungen übernommen, sofern es sich um feste Begriffe handelt.

Definition 2.1: Denotable Object
Denotable object ist der Sammelbegriff für alle im Modell auftretenden Datenelemente. Dies sind entweder *Literale* oder *Objekte*.

Definition 2.2: Literal, atomares/strukturiertes Literal, Datentyp
Ein *Literal* ist definiert als unveränderbarer Wert (*immutable object*) eines bestimmten *Datentyps*. Die *atomaren Literale* von ODMG gehören den üblichen Datentypen float, double, long, short, unsigned long, unsigned short, char, boolean und octet an. *Strukturierte Literale* werden aus anderen Literalen mittels der folgenden Typkonstruktoren gebildet: struct, union, enum, array, sequence, string, set, list und bag. Weiterhin können Felder fester Größe durch Anhängen einer Größenangabe in eckigen Klammern ([n]) definiert werden.

Die Datentypen sind sehr stark am C++-Typmodell orientiert — ein wesentlicher Unterschied ist allerdings, daß im ODMG-Datenmodell keine Zeiger zur Verfügung stehen. Präzision und Interpretation für die einzelnen Datentypen werden im Standard nicht vorgeschrieben, sondern der jeweiligen Implementierung überlassen[2].

Definition 2.3: Objekt, atomares/strukturiertes Objekt, Objektidentifier
Ein *Objekt* (*object, mutable object*) unterscheidet sich von einem Literal dadurch, daß es über einen sogenannten *Objektidentifier (oid)* identifiziert wird und seine Werte ändern kann, ohne seine Identität zu verlieren. Objekte beinhalten sowohl einen *Zustand* als auch ein zugehöriges *Verhalten*. *Strukturierte Objekte* sind strukturierte Literale mit zusätzlicher Objektidentität. *Atomare Objekte* werden hingegen über sogenannte Interface-Definitionen spezifiziert und gehören zu einem *Objekttyp* mit Struktur und Verhalten.

Literale erhalten ihre Identität aus dem dargestellten Wert. Dieser stammt aus einem vorgegebenen Wertebereich und kann sich folglich nicht ändern — ein anderer Wert ist stets gleichbedeutend mit einem anderen Literal. Dies gilt sowohl für atomare als auch für strukturierte Literale

2. Eine fragwürdige Entscheidung in Bezug auf Datenaustauschbarkeit; siehe Anhang G.

und entspricht der funktionalen, mathematischen Wertauffassung (vgl. [Day89]). Objekte hingegen behalten ihre Identität, auch wenn sich ihr Zustand ändert. Sie bleiben also „dasselbe“ Objekt, unabhängig von den in ihnen dargestellten Werten.

Anmerkung: Die Definition der strukturierten Objekte erscheint künstlich und unmotiviert. Ein solches Konzept ist bisher in keinem anderen Objektmodell vorgekommen und wird selbst innerhalb des ODMG–93-Standards nach seiner Definition nicht weiter verwendet. Es wird nicht klar, wie diese Verbindung zwischen Literaleigenschaft und Objektidentität realisiert werden soll, auch die Verwendung solcher Objekte wird nicht erläutert. Die Unveränderbarkeit der Literale und die Veränderbarkeit von Objekten stehen im Widerspruch zueinander. Die naheliegendste Interpretation ist die Verwendung strukturierter Objekte als Variablen, also als Speicherplatz für wechselnde Literale desselben Datentyps. Unabhängig davon ist die Bezeichnung der anderen Objekte als atomar ebenso irreführend, da in den Interface-Definitionen beliebig komplizierte Strukturen aufgebaut werden können. Aus diesem Grunde werde ich im folgenden strukturierte Objekte ignorieren und atomare Objekte schlicht als *Objekte* bezeichnen. Dies entspricht der üblichen Interpretation des Objektbegriffs und führt zu einer konsistenteren Begriffsbildung als im Standard selbst, in dem der Begriff Objekt je nach Kontext verschieden interpretiert wird.

Wie bereits erwähnt, gibt es im Datenmodell von ODMG keine Datenzeiger; Referenzen auf Objekte werden hingegen unterstützt. Im hier beschriebenen generischen Datenmodell wird jedes Auftreten eines Objekts stets als Referenz interpretiert. Dies entspricht der Objektauffassung von Smalltalk und steht im Einklang mit der Objektidentität, da jede Verwendung des Objekts stets auf den gleichen Speicherbereich verweisen kann. In der jeweiligen Sprachbindung wird diese Funktionalität durch die Einführung eines speziellen parametrisierten Referenzdatentyps erreicht. Dieser Datentyp implementiert das Konzept der sogenannten *Smart Pointer*, die im Gegensatz zu C++-Zeigern Wissen über die referenzierten Objekte haben und mehr Funktionalität anbieten können (siehe Kapitel 4.2.4 und 4.3).

Definition 2.4: Zustand, Datenfeld, Attribut, Relationship

Der *Zustand (state)* eines Objekts ist definiert durch eine Menge benannter *Datenfelder* (*properties)*. Datenfelder, die Literale enthalten, werden *Attribute* genannt, enthalten sie Referenzen auf Objekte, so nennt man sie *Pfade (traversal paths)*. Zu jedem Pfad muß auch ein inverser Pfad definiert sein. Ein solches Pfadpaar nennt man *Relationship*. Das Modell sichert *referentielle Integrität* auf diesen Relationships zu. Sie können dabei in unterschiedlichen Wertigkeiten auftreten (1:1, 1:n, n:m) und entweder geordnet oder ungeordnet sein (list, array bzw. bag, set).[3]

Referentielle Integrität bedeutet, daß beim Verändern einer Relationship stets automatisch beide Pfade korrigiert werden. Hierdurch unterscheiden sich Relationships von einfachen Zeigern, bei denen im Falle einer Wertänderung das referenzierte Objekt nicht über die Veränderung informiert wird. Obwohl der ODMG-Standard genau die obige Definition enthält, erlaubt er im Widerspruch dazu dennoch, Objektreferenzen in Attributen abzulegen. In diesem Fall verhält sich die Referenz wie ein Zeiger: Es gibt also keine Inverse, und referentielle Integrität wird nicht überprüft. Diese objektwertigen Attribute stellen die einzige Möglichkeit dar, Objektreferenzen innerhalb von Strukturen zu verwenden, da für Relationships nur einzelne oder mengenwertige Referenzen erlaubt sind. Weiterhin müssen Datenfelder zwar die Funktionalität eines Datenspei-

3. Im üblichen Sprachgebrauch bezeichnet man bereits die Pfade als Relationships. Der Begriff Pfad wird nur dann verwendet, wenn man eine einzelne Richtung der Relationship betonen will. Dem schließe ich mich im folgenden an.

chers bieten, können hierzu aber auch durch ein Funktionenpaar zum Lesen und Schreiben der Werte realisiert werden (get_value, set_value).

Definition 2.5: Verhalten, Operation, Methode, Ausnahme

Das *Verhalten (behaviour)* eines Objekts ist definiert als Menge seiner *Operationen* oder *Methoden*. Bei der Ausführung einer Operation auf einem Objekt wird eine *Parameterliste* übergeben und möglicherweise ein *Rückgabewert* als Ergebnis geliefert. Im Falle von Fehlern kann die Operation eine *Ausnahme (exception)* erzeugen, anstatt normal zurückzukehren.

Definition 2.6: Interface, Implementierung

Als *Interface* oder *Schnittstelle* eines Objekts bezeichnet man die Liste der Datenfelddefinitionen und Methodensignaturen des Objekts. Es definiert das von außen sichtbare Erscheinungsbild des Objekts. Eine *Implementierung* stellt die zugehörige Funktionalität zur Verfügung. Sie muß sowohl eine Speicherdarstellung der Datenfelder als auch Programmcode für alle Methoden enthalten. Für ein Interface können dabei parallel mehrere Implementierungen existieren.

Ein Interface ist direkt vergleichbar mit einer C++-Klassendeklaration. Es definiert alle Datenfelder und Methoden der Objekte. Diese Ähnlichkeit findet sich auch in der Spezifikationssprache ODL wieder (siehe Kapitel 2.1.2). Allerdings legt die C++-Klassendeklaration gleichzeitig die Repräsentation der Datenfelder und die Struktur der virtuellen Methodentabelle fest; sie enthält also zusätzlich bereits einen Teil der Implementierung.

Definition 2.7: Typ, Klasse, Extent

Ein *Typ (Objekttyp)* besteht aus einem Interface und der Menge aller zugehörigen Implementierungen. Er definiert die Struktur und das Verhalten all seiner Instanzen. Die Menge aller Instanzen eines Typs wird als der *Extent* des Typs bezeichnet. Die Instanzen können dabei von verschiedenen Implementierungen stammen. Als *Klasse* bezeichnet man die Zuordnung eines Interfaces zu einer konkreten Implementierung. Eine Klasse stellt also einen Ausschnitt des kompletten Typs dar.

Mit dieser Definition unterscheidet sich der ODMG-Standard von der meistens üblichen Interpretation von Typen und Klassen. Letztere interpretiert Typen stets als reine Strukturklassifikation (intensionaler Aspekt), während Klassen als Objektcontainer angesehen werden (extensionaler Aspekt). Die Zuordnung von Typen zu Klassen kann dabei mehr oder weniger Freiheiten erlauben. In den meisten Fällen werden jedoch nur 1-zu-1-Zuordnungen erlaubt, was die Unterscheidung zwischen den Begriffen meist überflüssig macht. Auch ich werde im folgenden auf die Unterscheidung zwischen Typ und Klasse verzichten, da die unten beschriebene C++-Sprachbindung ebenfalls nur eine Implementierung, und somit eine Klasse, für jeden Typ erlaubt.

Definition 2.8: Subtyp, Supertyp, Mehrfachvererbung, direkte/indirekte Instanz

Die Objekttypen in ODMG sind in einem Verband angeordnet. Die *Subtypen* erben die Struktur all ihrer *Supertypen*. Der Extent eines Subtyps ist stets eine Teilmenge des Extents der Supertypen. Ein Supertyp (Subtyp) T_1 von T_2 heißt *direkt*, wenn sich zwischen T_1 und T_2 keine weiteren Typen im Verband befinden. Ein Typ T kann mehrere direkte Supertypen haben (*Mehrfachvererbung, multiple inheritance*). Eine Instanz o eines Typs T heißt *direkte Instanz* von T, wenn sie in keinem Subtyp von T als Instanz erscheint. Anderenfalls heißt sie *indirekte Instanz* von T.[4] Jedes Objekt ist stets direkte Instanz genau eines Objekttyps. Dieser Typ wird bei der Erzeugung festgelegt und ist danach nicht mehr veränderbar.

Die Typhierarchie in ODMG ist explizit. Die Beziehung zwischen Objekttypen wird bei der Deklaration festgelegt, indem jedem Subtyp die Liste seiner direkten Supertypen zugeordnet wird. Bilden die Datenfelder und Methoden eines Typs rein strukturell eine Teilmenge der Datenfelder und Methoden eines anderen Typs, so folgt daraus nicht automatisch eine Supertyp-Beziehung. Gemäß der obigen Definition erlaubt ODMG also weder freie Typzugehörigkeit noch Objektmigration zur Laufzeit (siehe Abschnitt 2.2.4). Beide Eigenschaften werden jedoch als mögliche Erweiterungen des Standards in Betracht gezogen ([Cat94, S. 44]).

Definition 2.9: Metaschema: Type, Database, Transaction

Das *Metaschema* ist derjenige Ausschnitt aus dem Datenbankschema, der Schema-Beschreibungen modelliert. Die wesentlichen Elemente im ODMG-Metaschema sind die Typen Type, Database und Transaction. Die Instanzen von Type beschreiben die Struktur der in Definition 2.7 definierten Objekttypen, also die Interfaces. Die Instanzen von Database sind Stellvertreter für die jeweiligen ODMG-Datenbanken. Sie stellen unter anderem Funktionen zum Öffnen und Schließen der Datenbank zur Verfügung. Die Instanzen von Transaction definieren das in ODMG verwendete Transaktionsmodell.

Definition 2.10: Extentnamen, Objektnamen, Entry Points

Extents haben stets einen Namen. Einzelne Objekte können ebenfalls mit einer beliebigen Anzahl von Namen versehen werden. Diese Namen dienen als *Einstiegspunkte* (*entry points*) in die Datenbank; sie müssen innerhalb der Datenbank eindeutig sein.

In ODMG werden die Instanzen eines Typs nicht automatisch über den Typnamen angesprochen. Soll für einen Typ ein Extent verwaltet werden, so muß hierfür explizit ein Name vergeben werden. Die Namen von Objekten und Extents liegen im gleichen Namensraum, es gibt weder eine Typunterscheidung noch ein Namespace- oder Scoping-Konzept.

2.1.2 Die Definitionssprache ODL

Im Gegensatz zu den vordefinierten Literalen und strukturierten Objekten werden atomare Objekttypen (Interfaces) vom Benutzer definiert. Hierzu bietet der Standard eine deskriptive Definitionssprache, die *Object Definition Language (ODL)*. Diese Sprache basiert auf der *Interface Definition Language (IDL)* der Object Management Group ([OMG91]). Eine Schemadefinition besteht aus einer Liste von Definitionen von Interfaces, globalen Datentypen und Konstanten. Eine Interface-Definition hat das folgende Format:

4. Die Bezeichnungen für direkte und indirekte Instanzen unterscheiden sich in der Literatur. Byeon und McLeod verwenden die genannten Begriffe ([ByM93]), Abiteboul nennt die Instanzen *real* bzw. *virtual* ([Abi90]), und Bertino unterscheidet zwischen *instances* und *members* ([Ber92]). In vielen Fällen wird diese Unterscheidung überhaupt nicht getroffen.

```
interface mytype : supertype_list
    // type properties:
    ( extent extentname
      keys property_list ) : persistent
    {
    // instance properties:
    attribute       Type        attname;
    ...
    relationship    Othertype   relname  inverse Othertype::invrel;
    ...
    // instance methods:
                    ReturnType  methname ( in ParamType parameter ) raises ( Exception );
    ...
    };
```

In diesem Beispiel ist mytype der Name des Interfaces, und somit des Typs und der Klasse. Der neue Typ wird in der Typhierarchie als Subtyp der in supertype_list aufgeführten Typen eingetragen. Er erbt alle Datenfelder und Methoden dieser Supertypen. Die Auflösung von Namenskonflikten bei mehrfacher Vererbung wird im Standard nicht betrachtet. extentname definiert einen Entry Point, der alle Instanzen des Typs enthält. Das Schlüsselwort persistent besagt, daß einige dieser Instanzen persistent sein können. Die Entscheidung, ob eine bestimmte Instanz persistent ist oder nicht, wird für jedes einzelne Objekt zum Zeitpunkt der Objekterzeugung festgelegt. Ist der Typ als transient definiert, kann er keine persistenten Instanzen aufweisen.[5] Die Liste property_list nach dem Schlüsselwort keys definiert eine Liste von Datenfeldern, die einen Primärschlüssel für den Typ darstellen. Hinter keys kann wahlweise eine einzelne Schlüsselmenge, oder aber eine Liste alternativer Schlüsselmengen angegeben werden.

Der von geschweiften Klammern eingeschlossene Teil der Definition beinhaltet die eigentliche Strukturbeschreibung des Typs. Jede Attributdefinition wird durch attribute eingeleitet und bestimmt sowohl den Namen als auch den Wertebereich (*Domäne*, *Domain*) des Attributs. Relationship-Definition werden durch relationship eingeleitet und enthalten neben dem Namen und dem Zieltyp auch die Angabe der im Zieltyp definierten Inversen. Für die Methodensignaturen ist kein Schlüsselwort erforderlich. Im Gegensatz zu Methodendeklarationen in C++ wird jeder Parameter mit einer Kennzeichnung versehen, die angibt, in welcher Richtung der Datentransfer stattfindet. Parameter mit der Kennzeichnung in werden nur gelesen, out-Parameter werden erst innerhalb der Methode mit einem Wert versehen und inout-Parameter werden sowohl gelesen als auch verändert. Nach raises kann zusätzlich eine Liste der Ausnahmen angegeben werden, die innerhalb der Methode erzeugt werden können.

Der obige Code-Ausschnitt stellt nur ein Beispiel dar. Viele Elemente in der Interface-Spezifikation sind optional, und außerdem kommen noch Typ- und Konstantendefinitionen hinzu. Eine vollständige Grammatik findet sich in Anhang D. Neben der Verwendung zur Schemabeschreibung einer ODMG-Datenbank kann die Sprache auch als Austauschformat zwischen anderen Schemabeschreibungssprachen, wie etwa STEP/EXPRESS, SQL oder CFI verwendet werden ([Cat94, S. 50]).

Für die Spezifikation der Implementierung zu einem Interface stellt ODMG–93 keine eigene Sprache zur Verfügung. Implementierungen werden stets direkt in der Sprache programmiert, auf der die ODMG-Datenbank läuft (siehe Abschnitt 2.1.5, Sprachbindungen).

---

5. Der Standard läßt leider die Frage offen, was passiert, wenn transiente Typen als Sub- oder Supertypen persistenter Typen auftreten. Weiterhin ist die Angabe von persistent bzw. transient optional, ODMG definiert aber kein Standardverhalten für diesen Fall.

2.1.3 Die Anfragesprache OQL

Neben der Definitionssprache ODL definiert der Standard auch eine deklarative Anfragesprache, die *Object Query Language (OQL)*. Sie basiert auf der relationalen Anfragesprache SQL gemäß dem SQL–92-Standard ([DaD98]). Gerade im Zusammenhang mit den Standardisierungsbemühungen der OMG und der fortschreitenden Entwicklung im SQL-Bereich wird im ODMG-Standard bereits der Bezug zur neueren Version SQL3 aufgezeigt ([EiM99]), die größtenteils 1999 verabschiedet wurde. Das Ziel ist, OQL und SQL in Zukunft zu einer gemeinsamen Sprache zusammenfließen zu lassen. Die Spezifikation von OQL ist von O_2-Technology in den Standard eingebracht worden und entspricht im wesentlichen der Anfragesprache von O_2 ([BCD89], [BDK92]). Im Gegensatz zum ursprünglichen SQL zeichnet sich OQL durch eine bessere Orthogonalität und Abgeschlossenheit aus. Die Sprache enthält zwar noch das aus SQL bekannte „select-from-where“-Konstrukt, ist ansonsten aber als Algebra definiert. Jeder Ausdruck, der einen beliebig strukturierten Wert zurückliefert, gilt als gültige Anfrage und kann überall dort in andere Anfragen eingebettet werden, wo ein Wert des entsprechenden Typs erwartet wird. Viele Operatoren und Ausdrücke wurden dabei von C++ übernommen. Die folgenden Elemente können in einer OQL-Anfrage verwendet werden:

Elementarausdrücke:
Literale, benannte Konstanten, Variablen und Entry Points: 1, **true**, **nil**, ”John”, authors, ...

Konstruktoren:
Durch Anhängen einer Werteliste an einen Typnamen können strukturierte Literale und neue Objektinstanzen erzeugt werden:
set(1,3,7), **struct**(val:1, txt:”one”), Author(lname:”Busse”, fname:”Ralph”, publications:pubset), ...

Arithmetische Ausdrücke:
Mathematische Operatoren für atomare und strukturierte Literale:
not(**true**), 5 + 6, **list**(1,2) + **list**(3,4,5), set1 **union** set2, ...

Vergleiche und Quantoren:
Binäre Wertvergleiche sowie Existenz- und Allquantor zur Überprüfung einer Bedingung auf einer Menge von Werten. Die Operatoren **and** und **or** folgen dem sogenannten *shortcut*-Prinzip: Die Auswertung erfolgt von links nach rechts und wird abgebrochen sobald der Wert des Ausdrucks sicher feststeht.
forall p **in** parts: p.price>5, **exists** p **in** parts: p.price>10, ”yes” **in** answers, 5 <= 7, ...

Selektion:
Eine Verallgemeinerung des traditionellen „select-from-where“-Konstrukts aus SQL, bei der Methodenaufrufe, Objektausdrücke und Konstruktoren verwendet werden können und die Laufvariablen über beliebige Mengen iterieren. Der Ergebnistyp eines solchen Ausdrucks ist stets eine Menge oder Multimenge des durch die Projektionsklausel definierten Typs.

```
select distinct struct(Lage:s.Breitengrad, Temp: (m.Wert-32)*5/9)
        from s in Stationen, m in s.Messwerte
        where m.Skala = ”Fahrenheit”
```

Aggregatfunktionen:
Die aus SQL bekannten Aggregatfunktionen zur Gruppierung, Sortierung und Wertaggregation können in OQL auf beliebige Mengen angewandt werden:
sort by, **group by**, **count**, **max**, ...

Pfadausdrücke und Indexierung:
Mit Hilfe von eckigen Klammern können einzelne Elemente oder Abschnitte aus Listen

und Arrays angesprochen werden. Der Zugriff auf Strukturkomponenten, Datenfelder und Methoden erfolgt über Pfadausdrücke in Punkt- oder Pfeilnotation. Methodenparameter werden in runden Klammern an den Methodennamen angehängt.
list[1], aPart.id, myCompany→addEmployee(mable), ...

Konvertierungen:
Funktionen für die Umwandlung zwischen Listen, Mengen und flachen Werten sowie explizite Typisierung (*casting*):
element(**set**("singleton")), **listtoset**(**list**(1,2,3,4)), **flatten**(nested_collection), ...

Jede OQL-Anfrage kann mit **define** *identifier* **as** *query* mit einem Namen versehen werden. Dieser Name kann dann als Ausdruck in anderen Anfragen verwendet werden. In der Anfragesprache sind keine Update-Operationen vorgesehen, da der Standard davon ausgeht, daß alle Updates über Methoden ausgeführt werden. Die komplette Grammatik für OQL ist in Anhang E aufgeführt.

2.1.4 Die Manipulationssprache OML

ODMG–93 definiert keine explizite Manipulationssprache im generischen Teil des Standards. Durch die Objektorientierung sind viele Funktionen an Objektmethoden gebunden und stehen nicht als syntaktische Elemente einer eigenen Sprache zu Verfügung. Für jeden strukturierten Datentyp gibt es im Objektmodell eine solche Liste vordefinierter Methoden für diesen Typ. Zusammen mit den generischen Operatoren für atomare Datentypen (siehe OQL) definieren sie die zur Objektmanipulation notwendige Funktionalität. Die letztendlich anwendbare Manipulationssprache ist dagegen in der entsprechenden Sprachbindung festgelegt. Sie umfaßt die sprachspezifische Umsetzung der genannten Datentypen und erweitert sie um weitere in der Sprache zur Verfügung stehende Konstrukte.

2.1.5 Sprachbindungen

Der bisher beschriebene Teil des Standards definiert ein generisches Datenmodell mit den zugehörigen Sprachkonstrukten zur Definition, Abfrage und Manipulation. Um ausgehend von dieser Spezifikation eine reale ODMG-Datenbank zu implementieren, muß festgelegt werden, wie die generischen Konzepte in die jeweilige Programmiersprache abgebildet werden. Hierzu definiert ODMG–93 eine Reihe von Sprachbindungen, auf die ich im Implementierungskapitel (4.2.4) eingehen werde.

2.1.6 Anmerkungen zum Standard

Die Arbeit mit ODMG–93 hat sich als sehr schwierig erwiesen. Dies ist in erster Linie darauf zurückzuführen, daß sich der Standard noch immer in der Entwicklung befindet. Die ersten Veröffentlichungen waren zum großen Teil eine Sammlung mehr oder weniger weit fortgeschrittener Ideen, wie der Standard einmal aussehen soll. Besonders die erste Ausgabe 1.0 litt sehr darunter, daß die Kapitel von verschiedenen Autoren geschrieben wurden. Es fehlte an einer buchübergreifenden Korrektur, um die Inkonsistenzen und Widersprüche zwischen den einzelnen Kapiteln zu entfernen. Neben offensichtlichen Widersprüchen waren wesentliche Teile auch unklar beschrieben oder fehlten völlig (vgl. [Kay94], [Kim94a], [Kim94b], [CKZ94]). In Release 1.1 wurden einige Unstimmigkeiten behoben, andere wesentliche Fragen blieben unbeantwortet.

Um mit dem Standard zu arbeiten, war ich folglich gezwungen, Annahmen über die weitere Entwicklung zu treffen und eigene Interpretationen zu verwenden. Einige Kritikpunkte am Standard und die verwendeten Lösungen sind in Anhang G beschrieben. In Release 1.2 wurden viele dieser Probleme behoben, und es hat größere Design-Änderungen gegeben. Der in dieser Arbeit beschriebene Ansatz ist jedoch auf Basis von Release 1.1 implementiert worden, da Release 1.2 erst zu spät erschienen ist. Die notwendigen Anpassungen in Bezug auf neuere ODMG-Versionen werden gesondert in Kapitel 6.2 beschrieben.

2.2 Der Sichtbegriff

Der folgende Abschnitt beschreibt den Begriff der Sicht und zeigt seine unterschiedlichen Einsatzmöglichkeiten. Ausgehend von Sichten in relationalen Datenbanksystemen wird zu vergleichbaren Ansätzen im objektorientierten Bereich übergeleitet. Abschließend wird ein Überblick über das Aufgabengebiet der Datenbankintegration gegeben und der Zusammenhang mit Sichten erläutert.

2.2.1 Datenbankarchitektur: Sicht = externe Ebene

Der Begriff der *Sicht* (*View*) findet seinen Ursprung als *externe Ebene* im 3-Ebenen-Architekturmodell der ANSI/X3/SPARC Arbeitsgruppe ([ANSI75]). Sie stellt dort Mechanismen zur Verfügung, um das konzeptuelle globale Datenbankschema durch Auswahl, Einschränkung und Umstrukturierung an einzelne Anwender oder Benutzergruppen anzupassen (siehe Bild 1).

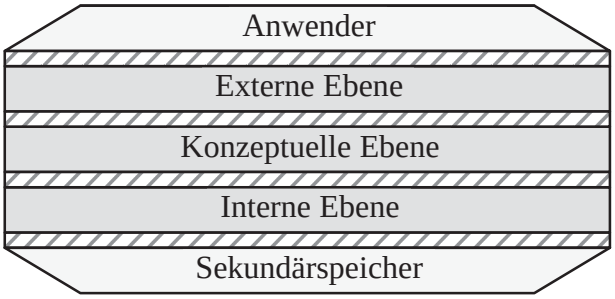


Bild 1: Das 3-Ebenen-Modell

Eine Sicht ist demnach ein möglicherweise strukturell verändertes Teilschema des konzeptuellen Schemas. Bietet ein System keine externe Ebene, so kann man das konzeptuelle Schema selbst als fixe „Universalsicht“ interpretieren, um das System in das 3-Ebenen-Modell einzuordnen ([Vos87]). Bietet ein System eine externe Ebene an, so kann auch das komplette Schema unverändert als Sicht auf sich selbst zur Verfügung gestellt werden. Dies erhöht die Zugriffssicherheit und erlaubt nachträgliche Schemaänderungen (*Schemaevolution*, siehe unten). Das einfache Prinzip der Einschränkung und Veränderung von Schemateilen in der externen Ebene kann dabei zu völlig verschiedenen Zwecken verwendet werden (siehe u.a. [Abi90], [Ber92], [DoE94]):

Datenanpassung: Jede Anwendung kann auf die Datenbank in genau der Form zugreifen, die für die jeweilige Aufgabe am besten geeignet ist. Man kann die Anwendung auf die eigentliche Funktionalität reduzieren und muß sich nicht um Konvertierungen kümmern.

Datenunabhängigkeit: Logische Datenunabhängigkeit tritt gleich in drei verschiedenen Varianten auf. Erstens sind die verschiedenen Anwendungen unabhängig voneinander. Jede bekommt ihr eigenes externes Schema, ohne durch die Anforderungen anderer Anwendungen beeinflußt

zu werden. Zweitens werden die Anwendungen unabhängig vom konzeptuellen Schema. Änderungen im konzeptuellen Schema können häufig durch Sichtanpassungen ausgeglichen werden. Die Anwendungen selbst können so auch mit der neuen Datenbank unverändert weiterarbeiten. Drittens ist auch die Datenbank selbst unabhängig von den Anwendungen. Der Schemaentwurf ist nicht mehr gezwungen, alle möglichen Forderungen der späteren Anwendungsprogramme zu berücksichtigen. Die letzten beiden Punkte gelten natürlich nur bis zu einem gewissen Grad. Eine radikale Änderung im konzeptuellen Schema kann auch durch Sichten nicht mehr ausgeglichen werden, und eine Fehleinschätzung im Entwurf kann die Leistungsfähigkeit (Performanz) der Anwendungsprogramme stark herabsetzen. Das allgemeine Konzept der Datenunabhängigkeit findet sich auch in den beiden konkreten Zielsetzungen Versionierung und Evolution (s. u.).

Zugriffssicherheit (Security): Der Zugriff auf vertrauliche Daten kann dadurch verhindert werden, daß die entsprechenden Teile des konzeptuellen Schemas nicht in die erlaubten Sichten der jeweiligen Benutzer übernommen werden (*Hiding*). Braucht ein Anwender für seine Arbeit dennoch Teile dieser Information, so kann gegebenenfalls durch strukturelle Umformungen eine Anonymisierung oder Akkumulation der Daten erreicht werden. Die Daten selbst sind also hinter den Zugriffsfunktionen verborgen (*Kapselung, Encapsulation*). Weiterhin können Sichten auch verwendet werden, um zwischen Lese- und Schreibrechten zu differenzieren.

Datensicherheit (Safety): Die Kapselung und die Unterscheidung von Lese- und Schreibrechten können auch helfen, unbeabsichtigte oder fehlerhafte Updates zu vermeiden.

Integrität: Eine spezielle Variante der Datensicherheit ist die automatische Konsistenzerhaltung ([MaM91]). Eine Update-Operation in einer Sicht kann so implementiert sein, daß sie sich automatisch um Integritätsbedingungen kümmert. Konfliktbehaftete Operationen können verweigert werden, notwendige Änderungen an mehreren Stellen im konzeptuellen Schema können von selbst angestoßen werden.

Versionierung: Besonders in objektorientierten Systemen besteht häufig der Bedarf, verschiedene *Versionen* eines einzelnen Objekts parallel zur Verfügung zu stellen. Neben dem *Rollenverhalten* ist hier speziell die zeitliche Varianz von Bedeutung. Bei Weiterentwicklungen einzelner Objekte kann es erforderlich sein, für vorhandene Programme auch weiterhin das alte Verhalten anzubieten. In Entwicklungsumgebungen ist es außerdem häufig notwendig, die Geschichte eines Objektes zu dokumentieren, oder erfolgte Veränderungen rückgängig machen zu können.

Schemaevolution: Im Gegensatz zur zeitlichen Entwicklung einzelner Objekte beschreibt die Schemaevolution die Veränderung des gesamten Schemas. Neben dem Problem der reinen Konvertierung der vorhandenen Daten, muß man auch die Anforderungen der vorhandenen Anwendungen beachten. Sollen die Anwendungen unverändert weiterbetrieben werden, kann man die jeweiligen Sichten an das neue Schema anpassen. Dies ist bereits unter Datenunabhängigkeit beschrieben worden. Es gibt aber auch noch zwei weitere Ansätze. Die erste Lösung ist, das neue globale Schema selbst als Sicht auf das bisherige konzeptuelle Schema zu definieren. Die alten Anwendungen arbeiten weiterhin mit dem alten Schema, alle neueren Entwicklungen basieren aber auf der neuen Globalsicht. Dieser Ansatz bricht jedoch mit der ursprünglichen Interpretation von Sichten als reine benutzerspezifische Anpassungen. Außerdem weicht es das Modell ein wenig auf, da im weiteren Verlauf die globale Sicht als eigentliches konzeptuelles Schema angesehen werden kann. Diese Aufweichung werden wir auch bei der Datenbankintegration wiederfinden. Die zweite Lösung verfolgt genau den umgekehrten Weg. Das konzeptuelle Schema und der gesamte Datenbestand werden physisch verändert, es wird aber zusätzlich eine globale Sicht definiert, die auf dem neuen Schema die alten Schnittstellen anbietet (siehe zum Beispiel [Sch93]). Die Sicht bildet jetzt quasi das konzeptuelle Schema für die bisherigen Anwendungen. Diese Varianten sind in Bild 2 noch einmal graphisch dargestellt.

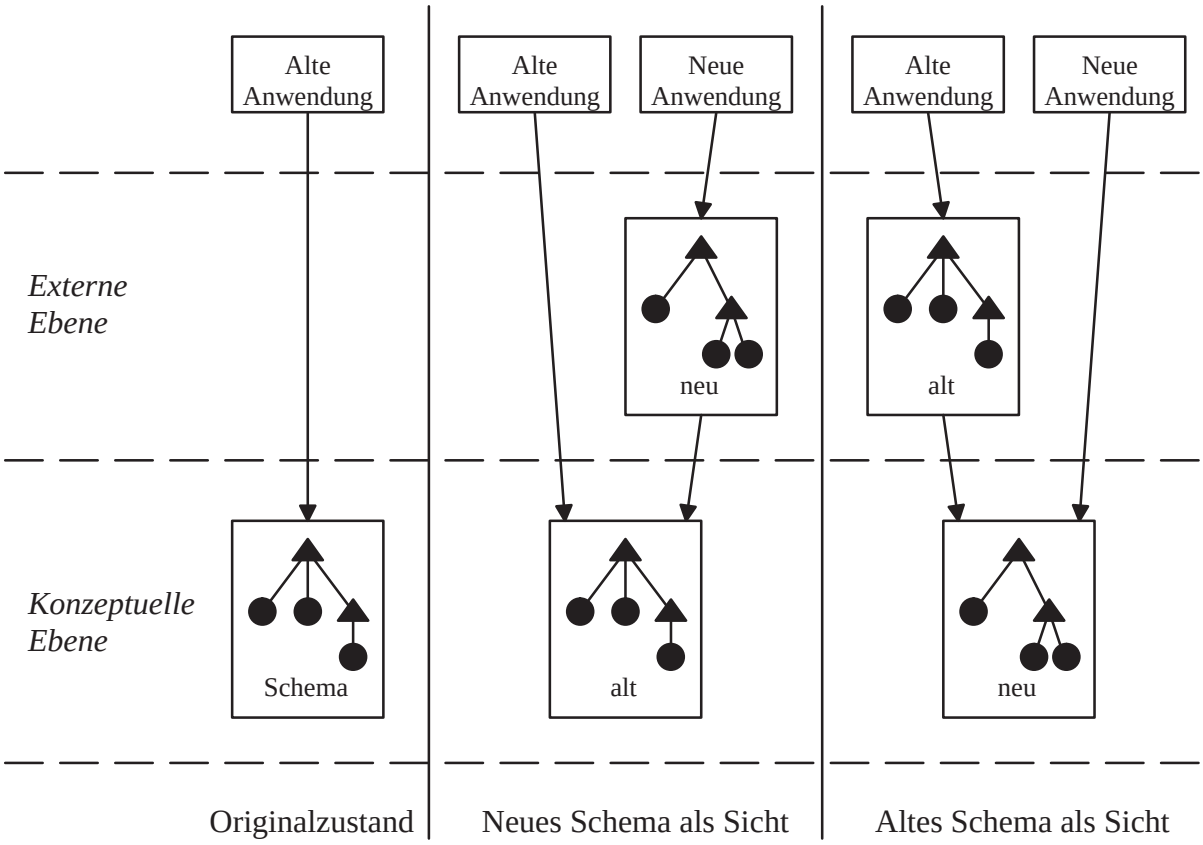


Bild 2: Sichteinsatz zur Schemaevolution

Arbeitserleichterung: Neben den inhaltlichen, funktionalen Aufgaben haben Sichten auch einen arbeitsergonomischen Aspekt. Durch die Anpassung des Schemas an den jeweiligen Benutzer wird dessen Umgang mit der Datenbank vereinfacht. Außerdem können im allgemeinen auch vordefinierte Anfragen für die tägliche Arbeit bereitgestellt werden.

Autonomie: Speziell bei der Datenbankintegration ist es häufig erwünscht, auf vorhandene autonome Datenbanken mit einem veränderten Schema zuzugreifen. Um die Autonomie der Originaldatenbank nicht zu verletzen, kann mit einer Sicht das neue Schema als Hülle um die Datenbank gelegt werden.

Datenabstraktion: Gerade im objektorientierten Bereich erlaubt die Kapselung durch eine Sicht, Interna des Schemas zu verstecken. Im Sinne des Konzepts der Datenabstraktion ist damit nur das nach außen dargestellte Verhalten von Bedeutung, nicht aber die letztendliche interne Realisierung ([HeZ90]). Dies ist aber weniger als Ziel der Sichtanwendung zu sehen, sondern eher als theoretische Klassifikation der bisherigen Punkte.

Die vorangegangene Liste vermittelt einen allgemeinen Eindruck von der Vielfalt der Anwendungsaspekte von Sichten. Die einzelnen Punkte lassen sich jedoch nur selten voneinander trennen. Einige der aufgeführten Einsatzgebiete überschneiden sich oder bedingen einander. Ihre Anwendbarkeit ist häufig auch durch das verwendete Datenmodell oder die jeweilige Systemimplementierung beschränkt. So ist etwa die Kapselung im relationalen Modell nicht möglich. Auch führen relationale Sichten eher zu Update-Problemen, als daß sie bei der Konsistenzerhaltung helfen.

2.2.2 Sichten im relationalen Modell

Im Bereich der relationalen Datenbanken ist der Begriff der Sicht sehr klar definiert. Ein Grund dafür ist sicher das sehr gute theoretische Fundament des relationalen Datenmodells. Durch die formale Faßbarkeit der Datenstrukturen und ihrer Operationen sind die Forderungen an Sichtimplementierungen klar zu beschreiben, und die Konsequenzen ihrer Benutzung lassen sich ebenfalls herleiten. Ein weiterer Grund ist die Abgeschlossenheit und Homogenität des Datenmodells, in dem sämtliche Daten als Tupel in Tabellen abgelegt sind. Das Ergebnis einer Sicht besteht folglich wieder aus einer oder mehreren Tabellen, die mit Daten aus dem Originalschema gefüllt werden. Zum Füllen dieser Tabellen verwendet man die ebenfalls genormte Anfragesprache SQL, die bereits Sichtdefinitionen in ihrer Syntax vorsieht.

Definition 2.11: Relationale Sicht, virtuelle Tabelle, Basistabelle
Eine *relationale Sicht* ist eine *virtuelle Tabelle*, die über eine Anfrage definiert ist. Bei Zugriff auf die virtuelle Tabelle wird diese Anfrage ausgeführt, um die Tupel der Tabelle zu berechnen. Die Tabellen, auf denen die Anfrage ausgewertet wird, werden als *Basistabellen* oder *Basisrelationen* der Sicht bezeichnet.

Die Tabellen heißen virtuell, weil sie nur während der Arbeit mit der Sicht existieren. Es wird nur der sie definierende Anfragetext in der Datenbank abgelegt, nicht aber die Tabelle selbst. Je nach Zugriffsverfahren muß eine solche Tabelle noch nicht einmal zur Laufzeit existieren — das System muß sich nur so verhalten, als ob es sie gäbe. Erscheint eine virtuelle Tabelle in einer Benutzeranfrage, so wird bei der Auswertung die Tabellendefinition so in die Anfrage eingebaut, daß das Ergebnis direkt aus den zugrunde liegenden realen Tabellen berechnet werden kann. Hierdurch erhält man beim Zugriff auf eine Sicht stets den aktuellen Datenbankinhalt — Änderungen auf den Basistabellen werden immer sofort sichtbar.

Im Gegensatz hierzu werden bei sogenannten *Snapshots* die Daten der Basistabelle zum Definitionszeitpunkt in eine temporäre Tabelle kopiert, auf der dann alle weiteren Zugriffe ablaufen. Änderungen auf den Basistabellen werden erst bei der Definition eines neuen Snapshots sichtbar. Ein wichtiges Anwendungsgebiet für Snapshots sind *Data Warehouses*, bei denen unternehmensweit Daten aus den verschiedenen Datenbanken gesammelt und zur weiteren Auswertung in einer umfassenden Datenbank abgelegt werden.

Updates

Die Umsetzung des Sichtkonzepts als benannte Anfrage funktioniert problemlos für rein lesenden Zugriff, da die Anfrage das Leseverhalten eindeutig definiert. Bei einem schreibenden Zugriff auf die virtuelle Tabelle treten jedoch Probleme auf, da das System anhand der Definition bestimmen muß, wie diese Schreibzugriffe auf die Basistabellen umgesetzt werden müssen. Dies ist in vielen Fällen nicht möglich, weil entweder wesentliche Daten für die notwendigen Operationen fehlen, oder aber die Operationen nicht eindeutig bestimmt werden können.[6] Nähere Untersuchungen zu Updates auf Sichten und der dabei auftretenden Seiteneffekte, sowie eine Klassifikation solcher Sichten findet man etwa in [Kel85], [Vos87, Kap. 23] oder [Dat86, Kap. 8.3].

6. Hier zeigt sich der wesentliche Unterschied zu Snapshots, bei denen Schreiboperationen direkt auf den kopierten Daten ausgeführt werden und die Basistabellen nicht beeinflussen.

2.2.3 Klassifikation objektorientierter Sichten

Genau wie im relationalen Fall ist eine objektorientierte Sicht eine umstrukturierte und teilweise eingeschränkte Darstellung des ursprünglichen Datenbankinhalts. Im Gegensatz zu relationalen Sichten können objektorientierte Sichten aber nicht immer als einfaches Anfrageergebnis verstanden werden. Die Reichhaltigkeit des objektorientierten Modells gibt mehr Möglichkeiten in der Ausdrucksfähigkeit von Sichten, erfordert aber gleichzeitig einen erhöhten Aufwand in der Einordnung der Ergebnisse in das existierende System, zum Beispiel in Bezug auf Objektidentität oder die Vererbungshierarchie. Allgemein kann man objektorientierte Sichten anhand der nachfolgenden Kriterien klassifizieren:

a) Verwendungszweck:
Das Ziel, das mit der Sicht erreicht werden soll.

b) Art der Transformation:
Die Eigenschaften der auf den Objekten auszuführenden Operationen.

c) Umgang mit Objektidentität:
Bleibt ein Bezug zwischen Originalobjekt und Sichtobjekt erhalten?

d) Art des Ergebnisses:
Typ und Granularität des Ergebnisses einer Sichtauswertung.

2.2.3.1 Verwendungszweck

Die allgemeine Klassifikation einer Sicht anhand ihres Verwendungszwecks (Datenanpassung, Sicherheit, Versionierung, usw.) gilt genau wie im relationalen Fall (siehe Abschnitt 2.2.1).

2.2.3.2 Art der Transformation

Eine andere Klassifikation unterscheidet die Art der Transformation, die auf den Objekten ausgeführt werden soll. Da eine Sicht das originale Schema mit einer veränderten Struktur und einem veränderten Umfang der Elemente zur Verfügung stellen soll, müssen die ursprünglichen Daten durch verschiedene Transformationen auf das gewünschte Endformat gebracht werden. In diesem Zusammenhang fassen wir die Transformation aber als *konzeptuelle Transformationen* auf. Wenn einem Objekt also zum Beispiel ein Attribut entfernt werden soll, weil dieses in der Sicht nicht mehr erscheint, so bedeutet dies nicht, daß die entsprechenden Attributwerte in den Datenbankobjekten physisch gelöscht werden. Eine solche Operation würde die Datenbank selbst verändern, was der Intention einer Sicht widerspricht. Das Ziel der Sicht ist es hingegen, Objekte zur Verfügung zu stellen, die sich so verhalten, als wäre aus den Original-Objekten das Attribut gelöscht worden. Je nach Implementierung kann dies etwa durch Kopier- oder Verdekkungsoperationen realisiert werden. Unabhängig von der letztendlichen Realisierung sprechen wir hier davon, daß das Originalobjekt zum Sichtobjekt transformiert wird. In diesem Sinne gibt es fünf grundlegende Arten von Objekttransformationen. Die ersten drei Arten verändern jeweils einen einzelnen Typ, die vierte beeinflußt mehrere Typen zugleich und die letzte führt keine strukturelle Transformation durch, sondern ändert nur einen Extent. Jede Sicht läßt sich als Folge solcher Basistransformationen darstellen.

Restriktion: Bei der Restriktion werden Attribute oder Methoden aus einem Objekt entfernt, bzw. unzugänglich gemacht (*information hiding*).

Konversion: Eine Konversion ist die strukturelle Veränderung der Objektdaten. Hierzu gehören Typkonvertierungen, wie etwa string↔integer, das Gruppieren einzelner Attribute zu Aggregaten, sowie Wertskalierungen ($↔Euro).

Augmentierung: Bei der Augmentierung werden einem Objekt neue Attribute oder Operationen hinzugefügt. Neben berechneten (abgeleiteten) Attributen können auch gewöhnliche Attribute hinzugefügt werden. Da es für sie jedoch keine Entsprechung im Originalschema gibt, können die darin abgelegten Daten eventuell nicht persistent gemacht werden.

Kombination: Werden Teile verschiedener Objekte zu einem neuen Objekt zusammengefügt, so spricht man von einer Kombination. Paare verheirateter Personen können in einer Sicht beispielsweise als einzelne Ehepaar-Objekte repräsentiert sein.

Selektion: Bei der Selektion werden nur bestimmte Objekte aus einer Objektmenge ausgewählt. Die Auswahl wird durch ein Prädikat, eine Bedingung, gesteuert. Zum Beispiel kann eine Sichtklasse `Expensive_Publication` definiert werden, die aus der Klasse `Publication` alle Instanzen auswählt, die die Bedingung `price > 100,00` erfüllen.

Kommerzielle Datenbanken, wie etwa Ontos, unterstützen häufig nur die Selektion sowie eingeschränkte Formen der anderen Transformationen, bei denen Objektinformationen verloren gehen. Forschungsprototypen bieten im allgemeinen die volle Mächtigkeit (siehe Kapitel 7). Das hier vorgestellte Sichtsystem unterstützt ebenfalls sämtliche Transformationen.

2.2.3.3 Art des Ergebnisses

Die obigen Transformationen legen nicht immer eindeutig fest, in welcher Form die einzelnen Ergebnisse zurückgegeben werden. Die ausgewählten Objekte können zum Beispiel auf ihre Attributwerte reduziert werden, was zu einem reinen Werte-Ergebnis führt. Umgekehrt kann man aber auch im Ergebnis auftauchende reine Werte als Attribute an neu generierte Objekte anhängen. Wenn eine Sicht Objekte liefert, muß weiterhin unterschieden werden, ob es sich dabei um die Originalobjekte der Datenbank oder um neu erzeugte Sichtobjekte handelt. Dieser Umgang mit Objektidentität stellt das wichtigste technische Unterscheidungsmerkmal der verschiedenen objektorientierten Sichtsysteme dar und bestimmt entscheidend dessen Funktionalität.

Definition 2.12: Wertbasierte Sicht
Eine Sicht, die ausschließlich Werte und keine Objekte zurückliefert, heißt *wertbasierte Sicht*. Dieser Ansatz entspricht dem relationalen Modell.

Definition 2.13: Objekterhaltende Sicht
Bei einer *objekterhaltenden Sicht (object preserving view)* werden die Originalobjekte der Datenbank, gegebenenfalls mit neuer Struktur, direkt in die Sicht übernommen. Sichtobjekte und Originalobjekte besitzen die gleiche Identität.

Definition 2.14: Objektgenerierende (objekterzeugende) Sicht
Eine *objektgenerierende (objekterzeugende) Sicht (object generating view)* erzeugt neue Objekte, die ursprünglich nicht in der Datenbank enthalten waren. Ein Sichtobjekt hat folglich eine andere Identität als das zugehörige Originalobjekt in der Datenbank.

Vom konzeptuellen Standpunkt ist fast immer der objekterhaltende Ansatz vorzuziehen, da ein Sichtobjekt im allgemeinen nur eine andere Darstellung desselben Realweltobjekts bedeutet. Außerdem wird die Konsistenzerhaltung erleichtert, da alle Änderungen — in der Originaldatenbank und in jeder beliebigen Sicht — stets auf demselben Objekt ausgeführt werden. Ein Beispiel hierfür ist die Definition einer Sicht als ein neuer Subtyp, so daß Datenbank und Sicht nur über verschiedene Interfaces innerhalb einer Typhierarchie auf dasselbe Objekt zugreifen können.

Viele der oben angegebenen Transformationen führen jedoch zu Strukturänderungen am Objekt, die nur unter bestimmten Bedingungen objekterhaltend durchgeführt werden können. Sie erfordern zum Beispiel freie Typzugehörigkeit von Objekten, dynamische Erweiterung des Typverbands oder Objektmigration, oder sie schränken die Datenbankautonomie ein (siehe Abschnitt 2.2.4). Ist eine objekterhaltende Sicht nicht möglich, so müssen neue Objekte erzeugt werden, die das veränderte Originalobjekt in der Sicht repräsentieren. Ein Realweltobjekt entspricht dann einer Äquivalenzklasse von Objekten, nämlich dem Originalobjekt und allen seinen Repräsentanten in den verschiedenen Sichten.

Manche Operationen können grundsätzlich nur objekterzeugend implementiert werden, da sie zu Objekten führen, zu denen es überhaupt keine (oder aber mehrere) Entsprechung(en) in der Datenbank gibt, wie zum Beispiel bei der Kombinations-Transformation.

Verallgemeinerte identitätserhaltende Sichten

Für die Beschreibung des von mir entwickelten Sichtmodells sind die Kategorien Objekterhaltend und Objektgenerierend nicht direkt anwendbar, da beide Charakteristiken auf verschiedenen Ebenen zum Einsatz kommen. Für eine genaue Charakterisierung des Modells werden daher die beiden folgenden Verallgemeinerungen definiert:

Definition 2.15: Identitätserhaltende Sicht

Eine *identitätserhaltende Sicht* ist eine Sicht, bei der sich eine eindeutige Zuordnung zwischen Originalobjekten und Sichtobjekten angeben läßt und diese dem System auch zur Verfügung steht. Für jedes Sichtobjekt ist das zugehörige Originalobjekt bestimmbar, so daß die Sichtobjekte aus semantischer Sicht dieselbe Identität aufweisen, auch wenn die technische Realisierung möglicherweise neue Objekte erzeugt hat.

Definition 2.16: Verallgemeinerte identitätserhaltende Sicht

Eine *verallgemeinerte identitätserhaltende Sicht* ist eine Sicht, die für all jene Objekte identitätserhaltend ist, die durch eine 1-zu-1-Abbildung aus Originalobjekten hergeleitet worden sind. Objekte, die durch Aggregation mehrerer Objekte oder aus Daten erzeugt worden sind, erhalten ebenfalls eine eindeutige Abbildung zu ihren Originalobjekten. Das Sichtobjekt stellt jedoch eine Art Gruppenidentität für die beteiligten Originalobjekte zur Verfügung.

Bei der Realisierung eines Sichtsystems für ODMG–93 auf Basis von C++ ist genau ein solches Konzept notwendig. Da beide Datenmodelle weder Migration noch freie Typzugehörigkeit zulassen, kann ein echt objekterhaltenden Sichtsystem nicht zur Verfügung gestellt werden. Die technische Realisierung wird folglich immer auf objektgenerierenden Mechanismen beruhen müssen. Dennoch ist es möglich, auf Basis dieser technisch objektgenerierenden Realisierung ein semantisch objekterhaltendes System zur Verfügung zu stellen, bei dem die neu erzeugten Objekte sich wie die Originalobjekte verhalten, nur daß sie sich in einer neuen Typumgebung befinden.

Das in den Kapiteln 3 und 4 vorgestellte Sichtsystem ist verallgemeinert identitätserhaltend.

2.2.3.4 Granularität und Einbettung

Der vorige Abschnitt hat untersucht, auf welche Art einzelne Objekte oder Werte zurückgegeben werden können. Diese Einzelergebnisse können darüber hinaus auch noch unterschiedlich gruppiert und in die existierenden Strukturen eingebettet werden. Je nach Integration in die Anwendungsumgebung und nach Abgeschlossenheit des Sichtsystems kann man die folgenden Fälle unterscheiden:

Datenmenge: Die Sicht liefert keine Objekte, sondern nur eine Menge strukturierter Daten zurück. Dies ist der Ergebnistyp für wertbasierte Sichten.

Objektmenge: Die Sicht liefert eine Menge von Objekten, die aber nicht an einen neuen Typ gebunden sind, sondern in einer einfachen Variablen abgelegt werden können. Die Sicht Expensive_Publication aus dem Selektions-Beispiel kann zum Beispiel in einer solchen Mengenvariablen abgelegt werden, da der Typ bei der Selektion nicht verändert wird.

Klasse:[7] Die Sicht liefert eine Menge von Objekten, die einem neuen Typ angehören. Dies ist zum Beispiel notwendig, wenn die Objektstruktur verändert worden ist, da die Sichtobjekte dann nicht mehr den Typdefinitionen der Originalobjekte genügen.

Schema: Liefert die Sicht eine ganze Hierarchie neuer Klassen, so bildet das Ergebnis ein eigenes Schema. Im Gegensatz zu reinen Klassensichten werden hierbei auch Relationships unterstützt.

Diese Varianten liefern Sichten mit steigender Funktionalität. Die Granularität Datenmenge entspricht dabei dem relationalen Fall: eine Sicht ist hier nur eine Menge flacher oder strukturierter Werte, die weder Objektidentität noch Verhalten aufweisen. Diese Art Anfragefunktionalität wird zum Beispiel von Ontos zur Verfügung gestellt, reicht für ein angemessenes Sichtsystem aber nicht aus ([ONT94c]).

Die Speicherung von Objektmengen in Variablen erlaubt die Ausführung reiner Selektionsanfragen ohne Umstrukturierungen. Diese Funktionalität ist typisch für objektorientierte Datenbanken, die keine strukturellen Objektveränderungen zur Laufzeit erlauben.

Durch die Erzeugung neuer Klassen zur Laufzeit können auch strukturelle Objektveränderungen vorgenommen werden. Der einfachste Fall sind hierbei *Subtyp-Sichten*, bei denen man durch Einfügen von Sub- oder Supertypen in den Typverband ein Objekt entlang der Vererbungshierarchie einschränken oder erweitern kann. Die verschiedenen Interfaces entlang der Vererbungskette eines Objekts — von der spezialisiertesten Klasse bis hin zur allgemeinsten Basisklasse — stellen so automatisch verschiedene Sichten auf dasselbe Objekt dar; die Objektidentität ist in allen Ausprägungen gewahrt. Hier zeigt sich eine Variation in der Auswertungsstrategie: Anstatt eine explizite Anfrage auszuwerten, wird eine neue Sichtklasse nur in den vorhandenen Klassenverband der Datenbank eingefügt. Die Instanzmenge ergibt sich dann automatisch aus der Inklusionsabhängigkeit zwischen den Klassen. Eine komplexere Version von Klassensichten erlaubt das Eintragen von Objekten an beliebige Stellen in der Klassenhierarchie; ein Objekt kann in diesem Fall gleichzeitig mehreren Typen angehören, die nicht in einer Sub- bzw. Supertypbeziehung zueinander stehen (siehe etwa [DoE94] oder [SLT91]). Ein solcher Ansatz stellt aber hohe Anforderungen an das Datenbanksystem, da das damit verbundene kontextabhängige Verhalten eines Objekts schwer realisierbar ist (vgl. [DoE96], [BeG95]). In vielen Fällen ist eine freie,

7. Bei einer Trennung von Klassen- und Typverband kann der Ergebnistyp „Objektmenge“ bereits als Klasse *ohne* neuen Typ dargestellt werden, während der Ergebnistyp „Klasse“ als Klasse *mit* neuem Typ aufgefaßt werden muß.

mehrfache Typzugehörigkeit einzelner Objekte nicht möglich. Strukturänderungen können dann nur durch die Erzeugung neuer Objekte realisiert werden (objektgenerierender Ansatz) — die Objektidentität geht in diesem Fall verloren.

Alle bisherigen Varianten haben immer nur einzelne Klassen verändert. Sobald Relationships zwischen diesen Klassen bestehen, müssen jedoch immer mehrere Klassen simultan in die Sicht übernommen werden, um referentielle Integrität bei Beziehungen zwischen Sichtobjekten zusichern zu können. Jede Sicht definiert also immer ein konsistentes Schema, das aber sowohl objekterhaltend als auch objektgenerierend verwendet werden kann.

Einbettung

Ein weiteres Detail bei Klassen- und Schemasichten ist die Frage der Einbettung in den ursprünglichen Typverband: neu erzeugte Typen und Klassen können in die existierenden Verbände eingebettet oder separat gehalten werden. Schemasichten bilden aufgrund ihrer Abgeschlossenheit üblicherweise eigene Verbände, während einzeln erzeugte Klassen sowohl eingebettet als auch separat gehalten werden können.

Das in dieser Arbeit vorgestellte Sichtsystem stellt vollständige Sichtschemata zur Verfügung, um eine maximale Modellierbarkeit des Datenbankschemas zu gewährleisten. Typen und Klassen der Sicht werden von denen der Datenbank getrennt.

Abhängigkeiten zwischen den Kriterien

Die vorgestellten Kriterien können im allgemeinen beliebig kombiniert werden, führen dabei aber zu unterschiedlichen Interpretationen des Ergebnisses. Wertbasierte Sichten können zum Beispiel beliebige strukturelle Änderungen vornehmen, verlieren aber jeglichen Zusammenhang mit den Originalobjekten und können nicht in ein Ergebnisschema eingebettet werden. Objekterhaltende Systeme unterstützen immer die reine Selektion; die anderen Transformationen erfordern Unterstützung durch hierarchische oder freie Typzugehörigkeit im System (siehe unten); die Kombination hingegen ist objekterhaltend überhaupt nicht ausführbar. Der objektgenerierende Ansatz erlaubt wie die Wertsichten beliebige strukturelle Änderungen. Die Ergebnisse können allerdings in eine Hierarchie eingebettet werden. Ob eine Sicht als einzelne Klasse oder als vollständiges Schema definiert wird, ist eher eine Frage von Abgeschlossenheit und Sichtbarkeit. Sowohl objekterhaltende als auch objektgenerierende Sichten können in beiden Formen spezifiziert werden. Tabelle 1 faßt die Möglichkeiten und Seiteneffekte der einzelnen Transformationen für die unterschiedlichen Formen der Objekterhaltung zusammen. Tabelle 2 stellt dar, ob Sichtobjekte in den vorhanden Klassen- bzw. Typverband eingebettet werden können.

	wertbasiert	**objektgenerierend**	**objekterhaltend**	**identitätserhaltend**
Restriktion	strukturell [1)]	Identitätsverlust [2)]	gut	gut [3)]
Konversion	strukturell [1)]	Identitätsverlust [2)]	gut	gut [3)]
Augmentierung	strukturell [1)]	Identitätsverlust [2)]	gut	gut [3)]
Kombination	strukturell [1)]	Identitätsverlust [2)]	nicht möglich	gut [3)]
Selektion	strukturell [1)]	Identitätsverlust [2)]	gut	gut [3)]

1) Die Operation liefert keine Objekte zurück.
2) Die Ergebnisobjekte haben eine andere Identität als die Originale; keine Updates, keine Einbettung in den alten Klassenverband.
3) Die semantische Identität wird vom System zugesichtert; Upfates sind möglich, keine Einbettung in den alten Klassenverband.

Tabelle 1: Transformationen und Objekterhaltung

	wertbasiert	objektgenerierend	objekterhaltend	identitätserhaltend
Datenmenge	lokal	n/a [1)]	n/a	n/a
Objektmenge	n/a	lokal	lokal	lokal
Klasse	n/a	separat	vorw. eingebettet	separat
Schema	n/a	separat	vorw. separat	separat

1) n/a = nicht anwendbar

Tabelle 2: Einbettung der Ergebnisobjekte

2.2.4 Datenbankeigenschaften: Typzugehörigkeit und Migration

Im vorigen Abschnitt wurde bereits erwähnt, daß bestimmte Formen von Sichten unterschiedliche Anforderungen an das Objektmodell der darunterliegenden Datenbank stellen. Im folgenden werden daher drei wichtige Eigenschaften objektorientierter Datenbanken erläutert, die einen wesentlichen Einfluß auf die Verfügbarkeit und Funktionalität objektorientierter Sichten haben. Sie bestimmen die Möglichkeiten eines Objekts, verschiedene Interfaces anzubieten.

2.2.4.1 Mehrfache Typzugehörigkeit (Polymorphismus)

Mehrfache Typzugehörigkeit beschreibt die Fähigkeit eines Objekts, gleichzeitig mehreren Typen bzw. Klassen anzugehören. Das Beispiel in Bild 3 beschreibt drei mögliche Stufen:

Einfache Typzugehörigkeit: Jedes Objekt gehört zu genau einem Typ, es gibt keinen Typverband.

Hierarchische Typzugehörigkeit: Jedes Objekt gehört zu genau einem spezifischsten Typ und allen seinen Supertypen (Vererbungshierarchie).

Freie Typzugehörigkeit: Jedes Objekt kann mehreren Typen angehören, die nicht in einer hierarchischen Beziehung zueinander stehen.

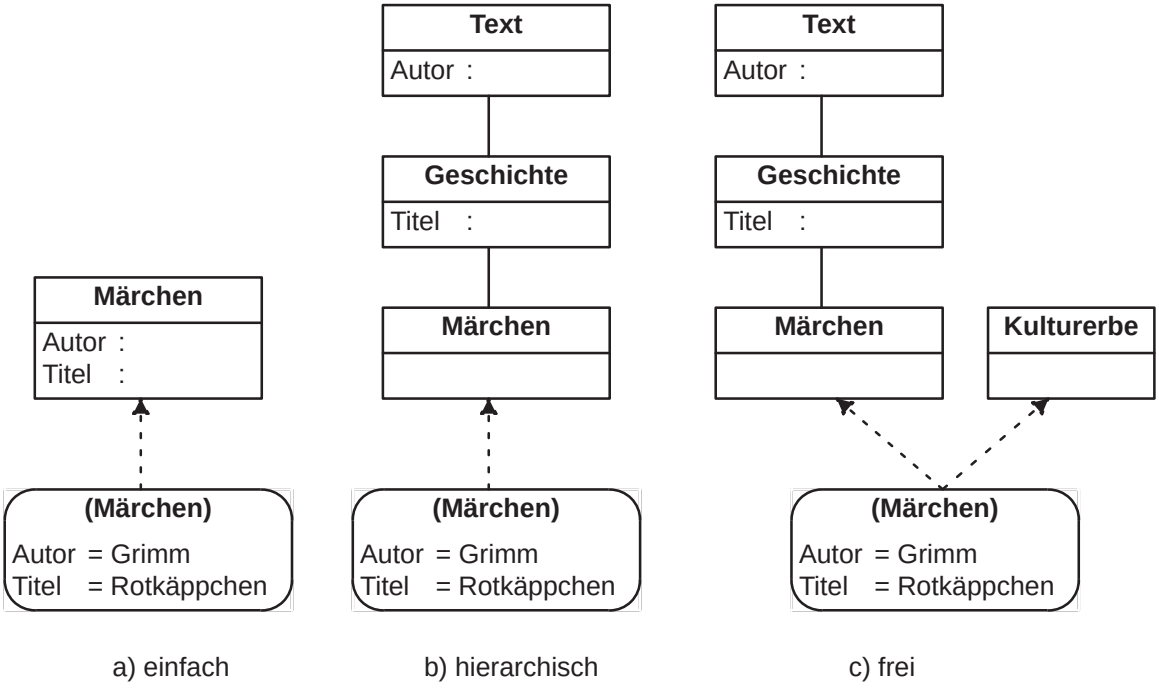


Bild 3: Typzugehörigkeit

Sowohl der ODMG-Standard, als auch die meisten derzeit verfügbaren Systeme bieten hierarchische Typzugehörigkeit. Einfache und freie Typzugehörigkeit sind eher ungewöhnlich. In den meisten Objektmodellen sind Typverband und Klassenverband miteinander gekoppelt: Alle Instanzen eines Typs werden in einer zugehörigen Klasse gesammelt und die Vererbungshierarchie im Typverband ist identisch mit der Inklusionshierarchie im Klassenverband. Statt von einfacher, hierarchischer und freier Typzugehörigkeit kann man also auch von einfacher, hierarchischer und freier Klassenzugehörigkeit sprechen. Bei einer Trennung von Typ- und Klassenverband muß man auch diese Begriffe voneinander trennen. Die Selektion aus Kapitel 2.2.3.2 beispielsweise, betrifft als extensionale Operation nur den Klassenverband, während die anderen Operationen die Objektstruktur ändern und somit auf dem Typverband operieren.

2.2.4.2 Objektmigration

Während mehrfache Typzugehörigkeit die grundsätzliche (statische) Fähigkeit eines Objekts beschreibt, zu mehreren Typen zu gehören, bedeutet Objektmigration das dynamische Ändern einer Objektstruktur zur Laufzeit. Auch hier kann man verschiedene Stufen unterscheiden (siehe Bild 4):

Keine Objektmigration: In diesem Fall wird die Klassen- und Typzugehörigkeit eines Objekts zu seinem Erzeugungszeitpunkt festgelegt und ist danach nicht mehr änderbar.

Aufwärtsmigration: Bei der Aufwärtsmigration oder generalisierenden Migration darf ein Objekt zur Laufzeit zu einem seiner Supertypen verschoben werden.

Hierarchische Migration: Bei hierarchischer Migration darf ein Objekt zur Laufzeit innerhalb der Vererbungshierarchie beliebig nach oben oder unten verschoben werden.

Freie Migration: Bei freier Migration kann ein Objekt zur Laufzeit jedem beliebigen Typ zugeordnet werden.

Die Migrationsfähigkeiten sind stark von den Typzugehörigkeitsfähigkeiten des Systems abhängig. Soll das Objekt bei der Migration seine alte Typzugehörigkeit nicht verlieren, so sind bei Aufwärtsmigration und hierarchischer Migration zumindest hierarchische Typzugehörigkeit notwendig, bei freier Migration auch freie Typzugehörigkeit. Die meisten heutigen Systeme bieten keine Migration, oder nur eine eingeschränkte Aufwärtsmigration an (z.B. Cast-Operatoren, die das Verhalten einer Subklasse ausblenden); die meisten Sichtansätze erfordern hingegen hierarchische oder freie Migration.

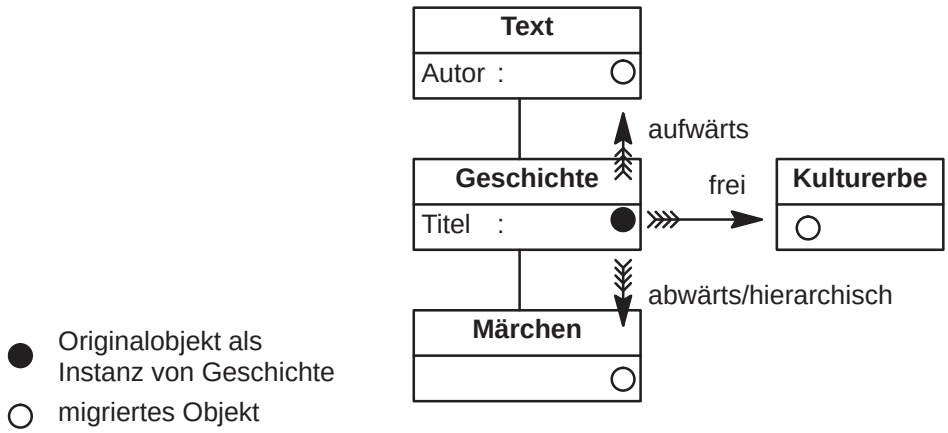


Bild 4: Objektmigration

2.2.4.3 Erweiterbarkeit des Schemas

Im Rahmen einer Klassen- oder Schemasicht werden neue Typen erzeugt, die noch nicht im Typverband vorliegen. Diese Typen können entweder eine unabhängige Hierarchie bilden oder in den vorhandenen Typverband eingeordnet werden. Im letzteren Fall kann man die folgenden Varianten unterscheiden (siehe Bild 5):

Eingeschränkte Erweiterbarkeit: Neue Typen (und Klassen) können nur als Blätter in die vorhandene Typhierarchie eingebettet werden. Ein neuer Typ kann also nie Supertyp eines bereits existierenden Typs sein. In Bild 5 ist dies die Fabel, die als neuer Subtyp unter dem Märchen eingefügt wird.

Volle Erweiterbarkeit: Neue Typen (und Klassen) können an beliebiger Stelle in der Originalhierarchie eingebettet werden. Vererbung und Inklusionsabhängigkeiten werden entsprechend angepaßt. Dies gilt in Bild 5 für den Text mit Titel, der zwischen Text und Geschichte eingefügt wird. Das Attribut Titel wird nun nicht mehr direkt in Geschichte definiert sondern von Text mit Titel geerbt.

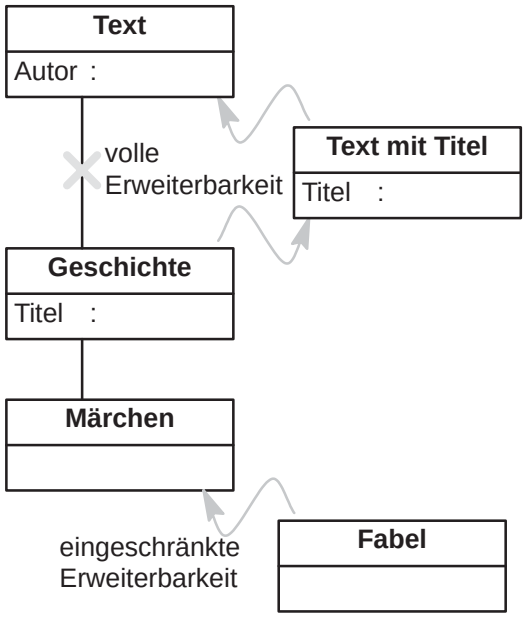


Bild 5: Erweiterbarkeit des Schemas

In heutigen Systemen steht üblicherweise nur eingeschränkte Erweiterbarkeit zur Verfügung; ein Beispiel für volle Erweiterbarkeit ist etwa MultiView ([KuR96a]).

Das in dieser Arbeit vorgestellte Sichtsystem stellt keine Anforderungen an Typzugehörigkeit, Migration und Erweiterbarkeit. Durch den identitätserhaltenden Ansatz werden intern stets neue Objekte erzeugt, so daß kein Objekt zu anderen Typen verschoben werden muß. Da diese neuen Objekte zudem in eine eigene Typhierarchie eingebettet werden, muß auch das Originalschema nicht erweitert werden.

Die folgende Tabelle gibt eine kurze Übersicht über die Anforderungen der im vorigen Abschnitt definierten Klassifikationskriterien. Der wertbasierte Ansatz stellt keine Anforderungen an das Datenbanksystem, da keine Objekte verändert werden und keine neuen Typen eingegliedert werden müssen. Der objektgenerierende und der identitätserhaltende Ansatz stellen ebenfalls keine Anforderungen an Typzugehörigkeit und Migrationsfähigkeit, da in beiden Fällen neue Objekte

erzeugt werden. Da die neu erzeugten Objekte jedoch zu neuen Klassen gehören, muß das Schema geeignet erweitert werden. Hierbei reicht hierarchische Erweiterbarkeit aus, da die neuen Klassen nicht in die existierenden Vererbungshierarchien eingebettet werden sondern einen neuen Verband bilden. Der objekterhaltende Ansatz ist der einzige, bei dem die existierenden Objekte dynamisch verändert werden müssen. Dies erfordert bei Restriktion und Augmentierung mindestens hierarchische Typzugehörigkeit und Migration, bei Konversion auf jeden Fall freie Typzugehörigkeit und Migration, sowie in allen Fällen volle Erweiterbarkeit. Nur die variablenbasierte Selektion ist hier anspruchslos, da keine strukturellen Änderungen vorgenommen werden.

	Typzugehörigkeit	**Migrationsfähigkeit**	**Erweiterbarkeit**
wertbasiert	keine	keine	keine
objektgenerierend	keine	keine	hierarchisch
objekterhaltend	hierarchisch / frei	hierarchisch / frei	voll
identitätserhaltend	keine	keine	hierarchisch

Tabelle 3: Systemanforderungen der Ansätze

2.2.5 Existierende objektorientierte Sichtmodelle

Durch das Fehlen eines einheitlichen Objektmodells existiert bisher auch kein einheitliches objektorientiertes Sichtmodell. Implementierungen objektorientierter Datenbanken reichen von persistenten Programmierumgebungen (ObjectStore, GemStone) bis zu verteilten Client-Server-Datenbanken mit deklarativem Zugriff, Transaktionen und Metaschema (O_2, COCOON, VODAK, usw.). Die Funktionalität der jeweiligen Datenbank hängt vom verwendeten Datenmodell und der darunterliegenden Programmiersprache ab. Beides beeinflußt auch die Anfragesprache und somit auch die Verfügbarkeit von Sichten. Bis auf wenige Prototypen gibt es meist überhaupt keine, oder nur sehr eingeschränkte Sichtunterstützung (vgl. [BaK90] und [Zdo93]). Im allgemeinen ist man auf reine objekterhaltende Selektionen oder auf umstrukturierte Wertemengen beschränkt (vgl. zum Beispiel ONTOS).

Die theoretischen Modelle hingegen gehen fast immer von einem optimalen System aus, das freie Typzugehörigkeit und freie Migration unterstützt. MultiView und COCOON zum Beispiel arbeiten rein objekterhaltend und ordnen die neuen Sichtklassen in den bereits vorhandenen Klassenverband wieder ein; die Modelle von Dobrovnik/Eder und dos Santos et al. verschieben sie in einen neuen Klassenverband. Der Ansatz von Heiler und Zdonik betrachtet als einziger objektgenerierende Sichten, während Chen/Shan einen völlig anderen Ansatz verfolgen und Objekte mit mengenwertigen Identifikatoren versehen, um direkt Gruppierungen oder Unifikationen darstellen zu können.

Eine detaillierte Übersicht über bereits existierende objektorientierte Sichten und Sichtmodelle wird in Kapitel 7 bei der Untersuchung verwandter Arbeiten gegeben. Diese Untersuchung zeigt einen großen Abstand zwischen „Theorie“ und „Wirklichkeit“: Keines der kommerziellen Systeme kann die Funktionalität bieten, die zur erfolgreichen Implementierung eines der theoretischen Sichtmodelle notwendig ist.

Aus diesem Grunde ist das hier vorliegende Sichtsystem entwickelt worden: Durch die Verwendung verallgemeinerter identitätserhaltender Sichten kann auf Basis eines realen Systems mit eingeschränkten Migrationsfähigkeiten ein semantisch reichhaltigeres Sichtsystem implementiert werden. Diese Sichten sind stets als vollständige Schemata definiert und unterstützen sämtliche Transformationen.

Updates

In Bezug auf Updates bietet die Mächtigkeit des objektorientierten Modells sowohl Vor- als auch Nachteile. Ein Vorteil ist sicherlich, daß sehr komplexe Updates mit Hilfe von Methoden explizit formuliert werden können. Ein Nachteil ist hingegen, daß durch die möglichen sehr komplexen Umstrukturierungen bei der Sichtdefinition im allgemeinen keine Inverse gefunden werden kann, die ein Sichtupdate in eine gleichwertige Operation auf den Originaldaten übersetzt. Immerhin wird bei objekterhaltenden Operationen zumindest zugesichert, daß jede Änderung direkt am Objekt vorgenommen wird — ob es nun direkt oder über eine Sicht bearbeitet wird (vgl. [Heu92, Kap. 6.3.5]).

2.2.6 Sichten in anderen Systemen

Bei objektrelationalen Datenbanken handelt es sich um relationale Datenbanken, bei denen zusätzlich komplexere Datentypen in den Tabellen abgelegt werden können. Die Sichten sind hierbei den rein relationalen Sichten gleichzustellen, da sie zwar die Erzeugung neuer virtueller Tabellen erlauben, aber keine spezielle Funktionalität zur Veränderung der eingebetteten Objekte bieten.

Bei den beiden „klassischen“ Datenbankmodellen, dem hierarchischen Modell (IBM, 1969) und dem Netzwerkmodell (CODASYL/DBTG, 1971), stehen ebenfalls Sichten zur Verfügung, die jedoch ausschließlich strukturelle Teilmengenfunktionalität bieten: man kann Schemateile umbenennen oder ausblenden und Zugriffsrechte einschränken, aber weder strukturell ändern noch wertbasiert einschränken ([Dat86], [LoS87], [Vos87]).

2.3 Sichteinsatz in interoperablen Datenbanken

Ein spezielles Anwendungsgebiet für Sichten liegt im Bereich des integrierten Zugriffs auf verschiedene Datenbanken. Für diese sogenannte Interoperabilität gibt es verschiedene Ausprägungen (vgl. [FNLS88], [ShL90]), bei denen Sichten mehr oder weniger stark eingesetzt werden können.

2.3.1 Interoperabilität

Definition 2.17: Interoperabilität

Unter *Interoperabilität* versteht man das Zusammenarbeiten mehrerer Datenbanken im Rahmen einer einzelnen Anwendung. Es wird hierbei nicht unterschieden, ob die Daten zwischen den Datenbanken ausgetauscht werden, oder ob sie gemeinsam einer Anwendung zur Verfügung stehen sollen.

Definition 2.18: Nicht-integrierte Interoperabilität, loosely-coupled federation

Unter *nicht-integrierter Interoperabilität* versteht man den expliziten direkten Zugriff auf die beteiligten Datenbanken. Jede Datenbank in dieser „schwach gekoppelten Föderation“ (*loosely-coupled federation*) wird mit ihrem Namen (ihrer Adresse) und ihrem eigenen Schema angesprochen.

Ein Beispiel für einen solchen nicht-integrierenden Ansatz ist die Anfragesprache MSQL, verwendet im relationalen System MRDSM ([GLRS93], [LAZN89]). Sie erweitert SQL um die Ar-

beit mit Tabellenmengen („semantic variables“) und erlaubt unvollständige Attributnamen. Die Ergebnisse auf allen beteiligten Datenbanken werden dann automatisch vereinigt, was bei unterschiedlichen Tabellen zu einer „Multitabelle“ im Ergebnis führen kann.

Beispiel 2.1:

Gegeben sei die Datenbank pubdb1 mit einer Tabelle autor und die Datenbank pubdb2 mit einer Tabelle author. Die folgende Anfrage sucht in beiden Tabellen nach Einträgen mit dem Namen „Busse“ und liefert die vereinigten Ergebnisse:

```
let db be pubdb1 pubdb2
select *
from db.aut%
where db.aut%.name = ”Busse”
```

■

Definition 2.19: Integrierte Interoperabilität, tightly-coupled federation

Bei *integrierter Interoperabilität* werden die einzelnen Datenbankschemata in ein gemeinsames Schema integriert. Die „eng gekoppelte Föderation“ (*tightly-coupled federation*) wirkt also nach außen wie eine eigenständige Datenbank. Die Zugriffe auf die eigentlichen föderierten Datenbanken bleiben dem Benutzer verborgen.

Beispiele für solche integrierten Datenbanken sind Systeme wie Multibase ([LaR82]) oder Mermaid ([TBC*87]), die Arbeiten von Spaccapietra und Parent ([SpP94]) sowie IRO-DB, in dessen Umgebung die vorliegende Arbeit entstand. Eine Gegenüberstellung von integrierendem und nicht-integrierendem Ansatz findet man in [FNLS88].

Beispiel 2.2:

In einer integrierten Datenbank sieht das obige Beispiel wie folgt aus:

```
select *
from all_authors
where name = ”Busse”
```

Bei der Definition der integrierten Datenbank wird bereits festgelegt, wie die globale Tabelle all_authors auf die beiden Originaltabellen pubdb1.autor und pubdb2.author abgebildet wird. Der Anwender muß sich um diese Details nicht mehr kümmern. ■

Beim integrierten Zugriff gibt es zusätzlich noch eine große Bandbreite beim Grad der Integration. Im einfachsten, aber unüblichen Fall besteht das integrierte Schema nur aus der disjunkten Vereinigung der Originalschemata. Die Schemaelemente (Tabellen, Klassen) existieren nebeneinander und halten ihre Instanzen streng getrennt. Diese Minimalintegration unterscheidet sich von der nicht-integrierten Interoperabilität nur durch das Verstecken der Verteilung. Man muß zwar nur noch ein Schema ansprechen, ist aber selbst für das Konvertieren und Zuordnen der Daten verantwortlich.

Bei maximaler Integration hingegen wird versucht, Korrespondenzen auf Schema- und Instanzebene zu finden und die korrespondierenden Elemente im integrierten Schema zu vereinigen. Auf Schemaebene bedeutet dies, daß Klassen oder Tabellen, die denselben Realwelt-Konzepten entsprechen, auf eine einzelne Klasse oder Tabelle abgebildet werden (Schemaintegration). Wird die integrierte Klasse (Tabelle) nun mit Instanzen gefüllt, so muß zusätzlich beachtet werden, daß einige Objekte (Tupel) der einen Datenbank und einige Objekte (Tupel) der anderen Datenbank denselben Realweltobjekten entsprechen. Diese dürfen in der integrierten Klasse

(Tabelle) nur einmal als Instanz erscheinen, während die korrespondenzfreien Objekte (Tupel) direkt übernommen oder ignoriert werden können (Datenbankintegration).[8]

Definition 2.20: Schemaintegration

Die *Schemaintegration* ist ein reiner Designprozeß. Sie beschäftigt sich nur mit dem Auffinden eines gemeinsamen Schemas für die gegebenen Ausgangsschemata. Implementierungsdetails und Instanzenkorrespondenzen werden hier außer Acht gelassen.

Definition 2.21: Datenbankintegration

Bei der *Datenbankintegration* wird eine neue Datenbank definiert, die auf vorhandenen Datenbanken aufsetzt. Zusätzlich zur Schemaintegration muß man dabei auf die externen Instanzen zugreifen und sie dem neuen Schema entsprechend konvertieren.

Schemaintegration wird unter anderem auch dort angewandt, wo es nicht um den interoperablen Zugriff auf existierende Datenbanken geht. Die Ausgangsschemata repräsentieren hierbei zum Beispiel alte Datenbanken, die komplett ersetzt werden sollen, oder Teilschemata aus verschiedenen Phasen im Designprozeß. Das generierte Schema wird dann als konzeptuelles Schema für eine neue Datenbank verwendet. Ein weiterer Unterschied zwischen Schemaintegration und Datenbankintegration ist die Veränderbarkeit der Ausgangsschemata. Bei der Schemaintegration ist es gelegentlich möglich, die Ausgangsschemata zu verändern, um zu einer optimalen Integration zu gelangen. Bei der Datenbankintegration wird dies durch die Autonomie der zugrunde liegenden Datenbanken verhindert.

[BLN86] gibt eine gute Übersicht über verschiedene zur Integration verwendete Verfahren, die Gemeinsamkeiten aus gegebenen Schemata extrahieren und dadurch zu einem umfassenden (minimalen) Schema gelangen. Eine neuere Arbeit in diesem Bereich ist die Dissertation von P. Fankhauser ([Fan97]), die ein Verfahren zur verlustfreien Integration vorstellt. Der dort vorgestellte Ansatz verfolgt — auf der Entwurfsebene — vergleichbare Ziele wie das hier beschriebene Sichtsystem. Die Umstrukturierung und Integration der Schemata erfolgt dort mit Hilfe dynamischer Augmentierungen, die die Mächtigkeit des objektorientierten Modells ausnutzen, um unter Erhalt der Klassenidentität zu einem vollständigen integrierten Schema zu gelangen. Die hier beschriebenen Sichten eignen sich somit sehr gut zur Realisierung dieses Integrationsansatzes, sind aber auch mit beliebigen anderen Verfahren kombinierbar.

2.3.2 Sichteinsatz in interoperablen Systemen

Bei der Realisierung eines interoperablen Systems können an verschiedenen Stellen Sichten eingesetzt werden. Bild 6 zeigt die unterschiedlichen Aufgaben, die dabei von Sichten übernommen werden können, und gibt eine Übersicht über mögliche Anwendungen mit unterschiedlichem Integrationsgrad.

Die Sichtaufgaben können in drei funktionale Ebenen eingeteilt werden: Import, Konvertierung und Vereinigung.

---

8. Das Übernehmen korrespondenzfreier Objekte in die Sicht entspricht dabei dem *Outer Join* ([Cod79]), bei dem die korrespondierenden Instanzen über einen Attributwert verbunden werden und die restlichen Instanzen beider Quellen mit Nullwerten ergänzt übernommen werden (siehe Kapitel 3.7.1).

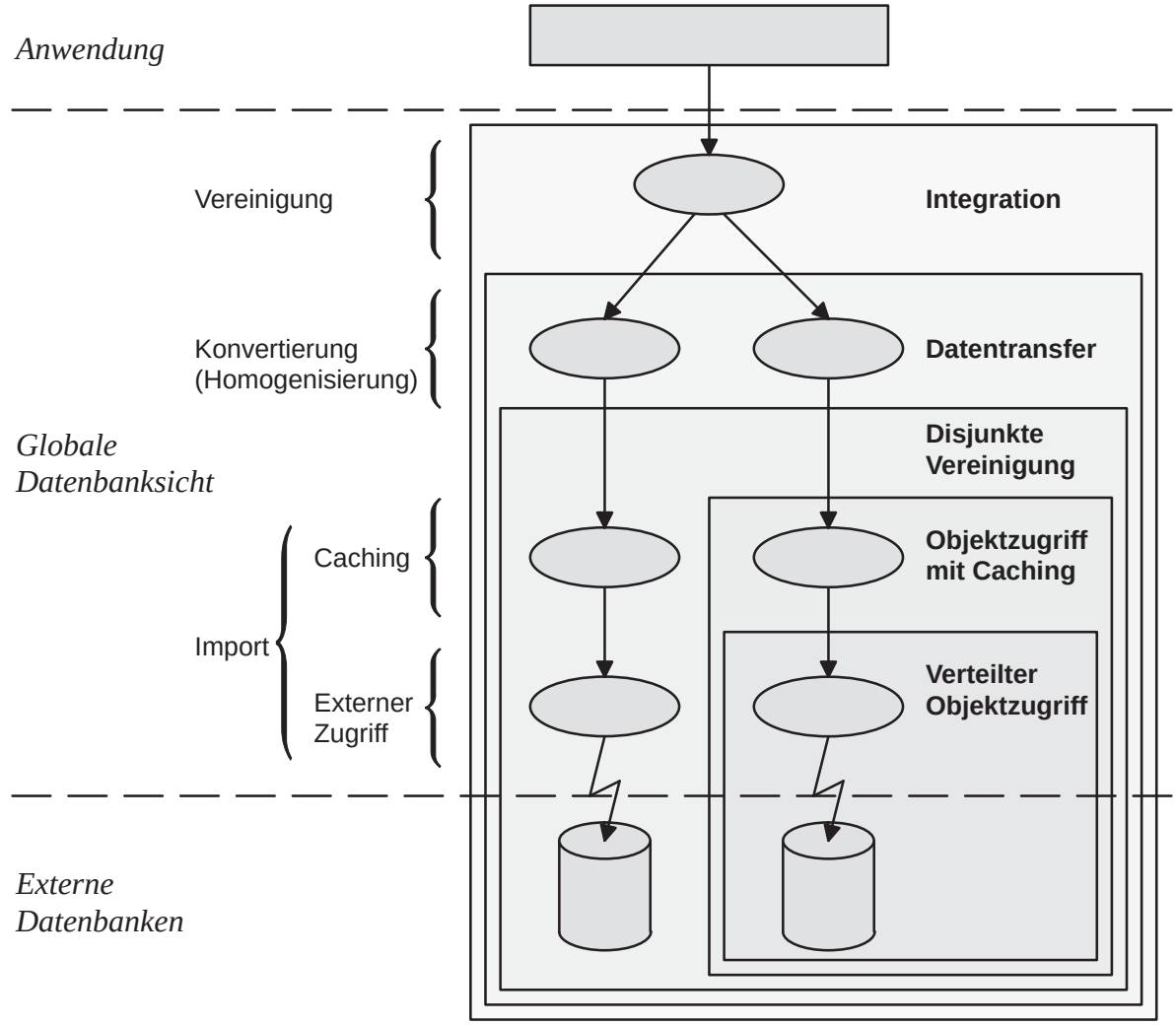


Bild 6: Variabler Integrationsgrad mit Hilfe von Sichten

Import: Das erste Einsatzgebiet für Sichten ist die reine Importfunktion, bei der die externen Daten global verfügbar gemacht werden.[9] Es findet dabei im allgemeinen keine strukturelle Veränderung statt — das Interface der Sicht ist fast immer identisch mit dem externen Original-Interface. Grundsätzlich sind nur zwei Modifikationen erlaubt: Um Namenskonflikte zu vermeiden, können Schemaelemente mit einem Präfix versehen oder geeignet umbenannt werden, und um die importierten Daten nicht zu umfangreich werden zu lassen, können unbenötigte Teile des Originalschemas ausgeblendet werden. Die Sicht wird dann angesprochen wie ein globales Schemaelement, bei dessen Auswertung dann automatisch auf die externe Datenbank zugegriffen wird, um die Instanzen zu holen. Die Verteilung, also die Verbindung zur externen Datenbank, ist dabei in der Sichtdefinition vor dem Anwender verborgen (*location transparency*). In diesem Importschritt kann man noch unterscheiden, ob die Sicht alle Zugriffe auf ihre Daten an die externen Datenbanken weiterleitet, oder ob die Daten in der Sicht zwischengespeichert werden, um die Zugriffe auf die externen Datenbanken zu reduzieren.

Konvertierung/Homogenisierung: In einer weiteren Schicht kann man die Struktur der importierten Daten verändern. Durch Konvertierungsklassen werden die Importklassen an die Anfor-

9. In der folgenden Diskussion bezeichne ich die integrierte Datenbank bzw. das Anwendungsprogramm als *global* und die föderierten Datenbanken als *extern* oder *lokal*.

derungen des Anwendungsprogramms angepaßt oder Daten verschiedener Quellen vereinheitlicht (*structural homogenization*). Im einfachsten Fall muß man nur Name und Typ einzelner Attribute ändern; eine Klasse in der Konvertierungsebene entspricht dann jeweils genau einer Klasse der Importebene. Ergänzend hierzu sind aber auch gelegentlich umfangreichere Umformungen vonnöten, bei denen Importklassen aufgeteilt oder kombiniert werden.[10] All diese Umformungen beziehen sich aber grundsätzlich nur auf Elemente einer externen Datenbank, also einer Importsicht.

Vereinigung: Sobald sich die Datenbestände der Datenbanken überlappen, muß man bei maximaler Integration noch die Schemaelemente und Instanzen miteinander vereinigen. Eine solche Sicht wird also Elemente aus verschiedenen importierten und konvertierten Sichten zu einem einzelnen Schemaelement zusammenfassen (*merge*). Im Gegensatz zu Konvertierungssichten, stammen hierbei aber die zu vereinigenden Daten aus verschiedenen externen Datenbanken. Es erfolgt also ein echtes Mischen der Datenbestände.

Durch die unterschiedlichen Aufgaben verwenden die einzelnen Schichten auch unterschiedliche Transformationen (siehe Tabelle 4). Die Importschicht übernimmt den eigentlichen Zugriff auf externe Objekte, während die anderen beiden dies nur indirekt tun. Selektionsfähigkeit wird auf allen Schichten gefordert, um die jeweils notwendigen Objektmengen für eine Transformation eingrenzen zu können. Darüber hinaus verwendet die Importschicht nur noch die Restriktion, um unnötige Teile der externen Datenbanken auszublenden. Die volle Bandbreite der Transformationen wird nur von der Homogenisierungsschicht benötigt, die alle strukturellen Änderungen vornimmt; Kombinationen werden dabei aber nur zwischen Instanzen jeweils einer Datenbank durchgeführt. Die Vereinigung benötigt dann nur noch die Kombination, um die bereits homogenisierten Klassen der einzelnen Datenbanken miteinander zu mischen.

Schicht	externe Objekte	Selektion	Restriktion	Konversion	Augment.	Kombinat.
Import	+	+	+	–	–	–
Homogenisierung	(–)	+	+	+	+	(+)
Vereinigung	(–)	+	–	–	–	+

Tabelle 4: Transformationen auf den einzelnen Ebenen

Diese funktionale Trennung der Sichten erlaubt eine optimale Anpassung an den geforderten Integrationsgrad. Mit reinen Import-Sichten kann man eine minimale Integration zur Verfügung stellen. Eine zusätzliche Konvertierungsschicht vereinfacht den Zugriff, da die Konvertierung nicht mehr in jeder Anwendung einzeln erfolgen muß. Durch Hinzufügen einer Vereinigungs-Schicht gelangt man dann zur maximalen Integration. Die folgenden Szenarien sind in Bild 6 markiert und demonstrieren sinnvolle Anwendungen der verschiedenen Integrationsgrade. Die ersten beiden Szenarien zeigen den Zugriff auf verteilte Objekte und arbeiten nur mit jeweils einer Datenquelle. Die folgenden beiden arbeiten auf mehreren, nicht integrierten Quellen, die sie als Multidatenbank ansprechen, und das letzte Szenario verwendet die vollständige Integration.

(a) Verteilter Objektzugriff mit direkter Propagierung

Durch die Bereitstellung von Sichtklassen für jede externe Klasse kann ein Anwendungsprogramm die externen Klassen wie lokale Klassen ansprechen. Die Verantwortung für die Vertei-

10. Eine gute Einführung in die Variationsvielfalt der Datenmodellierung, die bei einer Integration überwunden werden muß, gibt [Ken89].

lung ist nun dem System übertragen und obliegt nicht mehr der eigentlichen Anwendung, die dadurch einfacher und klarer implementiert werden kann. Im diesem einfachsten Fall wird auf Caching-Funktionalität verzichtet, so daß alle Objektzugriffe direkt an die externen Klassen weitergeleitet (propagiert) werden.

(b) Verteilter Objektzugriff mit Caching

Das zweite Anwendungsgebiet ergänzt die direkte Propagierung um Cache-Funktionalität. Durch Zwischenspeicherung von Objektwerten in der Importsicht kann die Netzbelastung verringert und die Effizienz beim Objektzugriff gesteigert werden. Die Zwischenspeicherung kann sich dabei wahlweise auf den kompletten Objektzustand oder auf einzelne Attribute beziehen. Der verteilte Objektzugriff mit Caching bietet sich vor allem dann an, wenn viele einzelne Lesezugriffe auf den importierten Objekten stattfinden. Auch Updates können beschleunigt werden, wenn alle Änderungen erst am Schluß der Transaktion gesammelt zur externen Datenbank übertragen werden. Bei der Verwendung externer Operationen kann die notwendige Invalidierung des Zwischenspeichers hingegen zu Performanz-Einbußen führen (siehe Kapitel 5). In diesem Fall kann direkte Propagierung bessere Ergebnisse liefern.

(c) Multidatenbank mit disjunkter Vereinigung

Importiert man Klassen von verschiedenen externen Datenbanken, so erhält man eine mehr oder weniger stark integrierte Multidatenbank. Bei der disjunkten Vereinigung werden die externen Klassen ohne weitere Bearbeitung direkt zur Verfügung gestellt. Die Klassen existieren mit disjunkten Instanzmengen nebeneinander. Es erfolgt keine Angleichung oder Mischung der Objekte. Eine solche Multidatenbank bietet noch keine Datenintegration, verbirgt aber die Verteilung durch Bereitstellung eines einzigen Gesamtschemas.

(d) Homogenisierter Datenzugriff und -transfer

Einen höheren Integrationsgrad als bei der disjunkten Vereinigung erhält man durch Aufsetzen einer zusätzlichen Schicht, in der die verschiedenen Klassen einander strukturell angeglichen werden. Obwohl die Instanzen noch immer getrennt sind, bietet eine solche Nachbearbeitung den Vorteil, daß der Datenaustausch zwischen dem Anwendungsprogramm und den einzelnen Datenbanken stark vereinfacht wird. Das Anwendungsprogramm ist nicht mehr für die notwendigen Konvertierungen zuständig; es kann mit denselben Datenstrukturen auf die Daten der verschiedenen Datenbanken zugreifen und so zum Beispiel leicht Daten 1-zu-1 von der Quell- zur Zielklasse übertragen; die notwendigen Anpassungen werden automatisch vom System übernommen.

(e) Integration

Der letzte Schritt führt zur eigentlichen Integration der Daten. Nach der strukturellen Angleichung werden nun auch die Objektmengen zusammengefaßt. Objekte verschiedener Datenbanken, die das gleiche Realweltobjekt darstellen, werden zu einem einzelnen Objekt verschmolzen und kommen in die gleiche Klasse wie Objekte, die nur in einzelnen Datenbanken vorkommen. Es hängt dabei vom Designer ab, wie vollständig diese Integration durchgeführt wird. Eine integrierte Datenbank kann man also als (mehrschichtige) Sicht auf den föderierten Datenbanken betrachten. Dies zeigt erneut die Ambivalenz des Sichtbegriffs: Aus Sicht der Integration stellt das integrierte Schema die externe Ebene der föderierten Datenbanken dar, aus Sicht eines Anwenders ist sie hingegen die konzeptuelle Ebene der integrierten Datenbank, auf der wiederum benutzerdefinierte Sichten angelegt werden können.

Die in Bild 6 verwendete Trennung der Integrationsphasen in explizite Schichten ist nur ein möglicher Ansatz. Genauso kann man die gesamte Funktionalität in einer einzelnen Sicht kombinieren, eine maximale Integration also zum Beispiel ohne explizites Importieren und Konvertieren definieren. Dies kann jedoch die Komplexität der Spezifikation erheblich erhöhen. Der Ansatz mit den getrennten Schichten entspricht der in einigen Integrationsverfahren verwendeten „Präintegrationsphase“, in der ebenfalls eine Homogenisierung der zu integrierenden Einzelschemata erfolgt, um den eigentlichen Integrationsschritt zu vereinfachen.

3 Entwurf und Spezifikation

Dieses Kapitel beginnt mit der Identifikation derjenigen Eigenschaften, die für ein universelles und interoperables Sichtsystem auf ODMG notwendig sind. Hieraus wird ein Modell abgeleitet, bei dem virtuelle Klassen mit Hilfe von Anfragen instanziiert werden und so die gewünschte Sichtfunktionalität zur Verfügung stellen. Auf einem technisch strikt objekterzeugenden System wird hierbei ein semantisch objekterhaltendes Verhalten zur Verfügung gestellt, bei dem die neu erzeugten Objekte eindeutig den Originalobjekten zuzuordnen sind. Es handelt sich also um ein verallgemeinertes identitätserhaltendes Sichtsystem. Im Anschluß an den Systementwurf wird die Syntax zur Spezifikation der virtuellen Klassen definiert. Details zu den Anwendungsmöglichkeiten der virtuellen Klassen und eine Diskussion über alternative und weiterführende Spezifikationen schließen das Kapitel.

3.1 Grundlegende Systemeigenschaften

In diesem Abschnitt bestimme ich eine Reihe von Anforderungen und Bedingungen für ein Sichtsystem auf Basis von ODMG–93, aus denen ich die folgenden acht grundlegenden Systemeigenschaften herleite. Die wichtigsten Entscheidungen betreffen dabei die Darstellung und die Berechnung der Sichtelemente:

1. Eine Sicht besteht nicht aus Werten sondern aus Objekten.
2. Die Objekte werden neu erzeugt.
3. Die Objekte bleiben über Rückverweise mit den Originalobjekten verbunden.
4. Die Rückverweise implizieren eine verallgemeinerte Identitätserhaltung.
5. Die Sichtklassen werden nicht in den ursprünglichen Typverband eingebettet.
6. Jede Sicht ist ein in sich abgeschlossenes Schema.
7. Sichtklassen werden durch mehrfache Anfragen definiert.
8. Sichtklassen können neue Methoden definieren.

Die ersten vier Eigenschaften definieren die Art des Ergebnisses gemäß Kapitel 2.2.3.3 und führen zum verallgemeinert identitätserhaltenden Ansatz. Die Eigenschaften 5 und 6 beschreiben Granularität und Einbettung gemäß Kapitel 2.2.3.4 und definieren die Sicht als ein vollständiges, nicht eingebettetes Schema. Die letzten beiden Eigenschaften zeigen schließlich die volle Sprachmächtigkeit dieses Ansatzes.

3.1.1 Art des Ergebnisses

a) Werte oder Objekte

Der relationale Ansatz, Sichten in Form einfacher SQL-Anfragen zur Verfügung zu stellen, kann nicht so leicht auf objektorientierte Datenbanken angewandt werden. Während im relationalen Fall die Daten nur aus flachen Werten bestehen, die beliebig zwischen Tabellen hin- und herkopiert werden können, hat man es im objektorientierten Fall mit Objekten zu tun, die sowohl Struktur als auch Verhalten aufweisen, wie es in den entsprechenden Klassendefinitionen festgelegt worden ist. Diese Definitionen können innerhalb einer Anfrage nicht verändert werden, da sich die Datendefinitionssprache von der Anfragesprache unterscheidet. Eine Anfrage bietet keine Möglichkeit, ein gegebenes Datenbankobjekt mit neuen Strukturelementen oder neuem Ver-

halten zu versehen. Man kann nur Daten extrahieren, aber nicht das Objekt selbst verändern. Darüber hinaus verliert man bereits beim reinen Kopieren eines Objekts dessen Objektidentität.

Solange man in der Sicht nur an den flachen Daten interessiert ist, kann man sie wie im relationalen Fall innerhalb der Anfrage beliebig umstrukturieren. Sobald man sie aus den Objekten extrahiert hat, besitzen sie keine Objektidentität und kein operationales Verhalten mehr. Das Ergebnis ist dann eine Menge strukturierter Werte, die keine Verbindung mehr zu ihren Originalobjekten aufweisen. Diese Situation ist aber in den meisten Fällen unerwünscht. Es ist ja gerade die Fähigkeit, Operationen auf den Datenbankobjekten auszuführen, die den wesentlichen Grund für den Einsatz objektorientierter Datenbanksysteme ausmacht. Außerdem unterstützen solche rein wertbasierte Sichten keine Updates, da die Verbindung zu den Datenbankinstanzen unterbrochen ist. Trotz dieser Nachteile und dem Bruch mit der Objektorientierung in der Ergebnisdarstellung sind solche Sichten durchaus in Einzelfällen von Bedeutung und sollten in der Form von benannten vordefinierten Anfragen vom System auch zur Verfügung gestellt werden. Dies ist bereits Teil des ODMG-Standards.

Für ein allgemeines Sichtsystem muß man jedoch Abgeschlossenheit in Bezug auf das Modell und volle Ausdrucksmächtigkeit im Ergebnis fordern. Hierzu gehört, daß das Ergebnis auch Objekte enthalten kann, die zudem strukturell verändert und mit anderem Verhalten versehen sein können. Jede Sicht muß folglich in Form von Klassen definiert werden.

Eigenschaft 1

Um ausreichende Funktionalität zu bieten und neue Objekttypen definieren zu können, besteht jede Sicht aus Klassen und Objekten.

b) Objektidentität

Die nächste Entscheidung ist, ob die Sichtklassen mit bereits vorhandenen oder mit neu erzeugten Objekten gefüllt werden sollen, ob man also ein objekterhaltendes oder ein objektgenerierendes System zur Verfügung stellen will. Wie in Abschnitt 2.2.3.3 erläutert, werden im objekterhaltenden Ansatz die Originalobjekte aus der Datenbank direkt mit den neuen Eigenschaften versehen. Die Instanzen der Sicht haben also die gleiche Objektidentität wie die ihnen zugrunde liegenden Originalobjekte. Aus theoretischer Sicht ist dieser Ansatz der ‚natürliche', da man ja nur eine neue Sichtweise auf die alten Objekte haben will, also ‚eigentlich' mit den ursprünglichen Objekten weiterarbeitet. Außerdem bietet dieses Verfahren die automatische Zuordnung von Sichtobjekten zu persistenten Datenbankobjekten, was sowohl für die inkrementelle Sichtberechnung als auch für das Updateverhalten ein wesentlicher Vorteil ist. Aus diesem Grunde basieren die meisten existierenden Ansätze für objektorientierte Sichten auf diesem Prinzip (siehe Kapitel 7).

So schön dieses Verfahren vom theoretischen Gesichtspunkt auch sein mag, so ungünstig sind aber die Bedingungen für eine reale Implementierung. Da bei diesem Verfahren den existierenden Objekten zur Laufzeit ein neuer Typ zugeordnet wird, muß das System sowohl Objektmigration als auch mehrfache Typzugehörigkeit unterstützen. Die Datenbankobjekte müssen gleichzeitig sowohl das Interface ihres Ursprungstyps, als auch das des neuen Sichttyps unterstützen. Diese Eigenschaften werden aber weder vom allgemeinen ODMG-Objektmodell noch von dessen C++-Sprachbindung unterstützt ([Cat94, S. 16]). Zwar werden diese Eigenschaften als mögliche Erweiterungen in zukünftigen Versionen des Standards aufgeführt (vgl. [Cat94, S. 44]), aber bisher hat noch keine entsprechende Änderung stattgefunden. Solange ODMG–93 die direkte C++-Abbildung beibehält, wird eine solche Änderung auch immer auf das generische Modell beschränkt und nicht in der Sprachbindung durchführbar sein. Auf Grund dieser Einschrän-

kung muß bis auf weiteres jedes Sichtsystem für ODMG–93, das nicht den Standard selbst verändert, auf dem objektgenerierenden Ansatz beruhen, bei dem für jede Sicht ein neues Objekt als Stellvertreter für das Originalobjekt zur Verfügung gestellt wird.

Abgesehen von diesem technischen Problem gibt es auch funktionale Gründe, die für einen objektgenerierenden Ansatz sprechen:

- Zugriff auf externe Objekte

  Um die Sichten zur Datenbankintegration und -interoperabilität einsetzen zu können, muß man auch auf externe Objekte zugreifen können. Da sich diese Objekte nicht im Objektraum der Sicht befinden, müssen Proxy-Objekte angelegt werden, die als Stellvertreter der externen Objekte in der Sicht fungieren.

- Unifikation von Objekten

  Speziell bei der Datenbankintegration, aber auch bei anderen vereinigenden Sichten, stellen verschiedene Objekte häufig dasselbe Realweltobjekt dar. In einer Sicht ist es deshalb oft wünschenswert, solche korrespondierenden Objekte zu einem einzelnen Objekt zusammenzufassen. Das neue Objekt kann dann allerdings nicht die Identität aller Originalobjekte übernehmen, da die Objektidentität ja eindeutig sein muß.[11] Die Verwendung einer ausgewählten OID aus der Menge der Originalobjekte führt dagegen zu Problemen bei der Zuordnung aller anderen Objekte. Im allgemeinen ist es deshalb einfacher, für das integrierte Objekt ein neues Objekt mit eigener OID zu erzeugen.

- Objektgruppierungen

  Eine andere Variante beim Zusammenfassen von Objekten ist das Bilden neuer Gruppierungen (*Aggregation*). Hierbei werden keine korrespondierenden Objekte unifiziert, sondern sich ergänzende Objekte zu einem neuen Konzept zusammengefaßt. Ein traditionelles Beispiel hierfür ist die Generierung eines Ehepaar-Objekts aus zwei Personen-Objekten.

- Objektifizierung von Daten

  Neben der Wiederverwendung und Umstrukturierung von Datenbankobjekten kann es auch notwendig sein, ursprünglich literale Daten in Form von Objekten zur Verfügung zu stellen. Dies ist zum Beispiel der Fall, wenn Daten integriert werden sollen, die in einer Datenbank als Objekte und in einer anderen Datenbank als Literale modelliert sind.

Aus all diesen Gründen heraus beruht das in dieser Arbeit beschriebene Sichtsystem auf dem objektgenerierenden Ansatz. Die zugrunde liegende Idee für objektgenerierende Sichten stammt von Heiler und Zdonik ([HeZ90]). Sie ist in dieser Arbeit konkretisiert, erweitert und auf C++ und ODMG–93 übertragen worden.

Eigenschaft 2
Es werden keine Objekte aus der Datenbank direkt in Sichtklassen übernommen. Alle Objekte der Sicht sind neu erzeugt.

Es gibt zwar auch weiterhin benannte Anfragen, die Werte oder existierende Objekte zurückliefern können, diese werden im folgenden aber nicht weiter betrachtet, da sie bereits Teil von ODMG–93 sind und keine besondere Sichtfunktionalität benötigen.

---

11. Die mengenwertigen OIDs von [ChS93] stellen hierbei eine mögliche Alternative dar. Mengenwertige OIDs bedeuten allerdings einen erhöhten Verwaltungsaufwand und werden meines Wissens bisher von keinem System unterstützt.

c) Objektkorrespondenz durch Rückverweise

Im objekterhaltenden Ansatz werden die gleichen Objekte sowohl in den Originalklassen als auch in den Sichtklassen verwendet. Jedes Objekt kann also direkt in den verschiedenen Ausprägungen auftreten, bleibt aber aufgrund der Objektidentität *dasselbe* Objekt. Ein Update auf dem Objekt ist folglich in allen seinen Ausprägungen gleichzeitig sichtbar, da das Objekt selbst verändert wird. Im objektgenerierenden Fall gilt dies nicht. Zu dem Zeitpunkt, zu dem die Sicht berechnet wird, werden die Sichtklassen mit neu erzeugten Objekten gefüllt. Diese sind also, technisch betrachtet, *andere* Objekte als die Originale. Es muß folglich explizit eine Zuordnung zwischen Sichtobjekten und Originalobjekten verwaltet werden, um Updates auf Originalobjekten in der Sicht sichtbar zu machen, und um Änderungen an einem Sichtobjekt an das Originalobjekt weiterleiten zu können. Bei einem Lese- oder Schreibzugriff auf das Sichtobjekt kann dann mit Hilfe der Zuordnung das entsprechende Originalobjekt bestimmt und der korrespondierende Zugriff auf diesem Objekt durchgeführt werden (siehe zum Beispiel [Ber92], [HeZ90], [YoC93]).

Bei der Realisierung dieser Zuordnung muß man zwischen globalem und lokalem Ansatz unterscheiden. Bei der globalen Lösung werden alle korrespondierenden Objekte in eine oder mehrere Systemtabellen eingetragen. Die Zuordnung erfolgt dann durch Nachschlagen in diesen Tabellen. Bei der lokalen Lösung hingegen wird in jedem Sichtobjekt selbst eine Referenz auf das zugehörige Originalobjekt abgelegt (Rückverweis). Dadurch kann man direkt vom Sichtobjekt zum Originalobjekt navigieren. [Ber92] bietet zum Zugriff auf die Originalobjekte eine Systemfunktion baseobject an. Da es sich um eine generische Funktion handelt, liegt der globale Ansatz nahe. In meinem System wird hingegen die lokale Lösung verwendet, da sie zwei wesentliche Vorteile bietet: Erstens ist sie flexibler als eine tabellarische 1-zu-1-Zuordnung, da man die Referenz an den Bedarf jeder einzelnen Klasse anpassen kann. So ist es zum Beispiel möglich, auf mehrere Originalobjekte zu verweisen oder zusätzliche Schlüsselwerte zur Referenz hinzuzufügen. Zweitens erlaubt die Referenz eine sehr elegante Spezifikation der virtuellen Attribute und Relationships, da man in der Ableitungsklausel leicht über die Referenz zum Originalobjekt navigieren kann (siehe Abschnitt 3.4).

Eigenschaft 3

> Um Updates zu ermöglichen, muß man beim objektgenerierenden Ansatz eine Zuordnung zwischen Sichtobjekten und Originalobjekten verwalten. In dem hier vorgestellten System erfolgt dies durch Rückverweise in den Sichtobjekten.

Diese Rückverweise erlauben nur die Navigation vom Sichtobjekt zum Originalobjekt. Eine gegebenenfalls notwendige inverse Zuordnung muß in Systemtabellen verwaltet werden, da aufgrund der Autonomie die Ablage von Verweisen in den Originalobjekten nicht erlaubt ist, und die Iteration über alle Sichtobjekte zu ineffizient wäre.

d) Semantische Identitätserhaltung

Eine wesentliche Eigenschaft objektorientierter Systeme ist die Objektidentität. Üblicherweise erhält jedes Objekt zum Zeitpunkt seiner Erzeugung eine OID, die dieses Objekt eindeutig kennzeichnet. Jedes neu erzeugte Objekt erhält automatisch seine eigene, neue OID. Im Falle von Sichtobjekten kann man dieses einfache Verfahren nur bedingt anwenden. Die Tatsache, daß Sichtobjekte nicht ‚von sich aus' existieren, sondern in ihrer Existenz von ihren Basisobjekten abhängen, bedingt eine Abhängigkeit zwischen der Objektidentität des Sichtobjekts und der seiner Basisobjekte (*semantische Identitätserhaltung*). Bei eins-zu-eins-Abbildungen übernimmt das Sichtobjekt quasi die Identität des Originalobjekts im Rahmen der Sicht. Bei n-zu-eins-Ab-

bildungen repräsentiert das Sichtobjekt alle beteiligten Originalobjekte und übernimmt die Identität der so definierten Äquivalenzklasse. Die Objektidentität eines Sichtobjekts kann folglich aus seinen Rückverweisen bestimmt werden. Wird innerhalb einer Datenbanksitzung zum ersten Mal auf ein Sichtobjekt zugegriffen, so wird dieses Sichtobjekt neu erzeugt und erhält eine eigene OID. Diese OID ist für den Rest der Datenbanksitzung gültig (*temporäre Objektidentität*). Bei erneutem Zugriff auf die Sichtklasse muß daher für jedes neu zu erzeugende Sichtobjekt überprüft werden, ob es bereits ein entsprechendes Sichtobjekt für die jeweiligen Basisobjekte gibt. Existiert ein solches bereits, muß die Objekterzeugung ignoriert und statt dessen das bereits vorhandene Objekt wiederverwendet werden. Diese Zusicherung der Eindeutigkeit kann dabei wahlweise vor oder nach der Objekterzeugung geschehen: [AbB91] zum Beispiel verwendet spezielle Konstruktoren für *imaginary objects*, die vor der Objekterzeugung nach einem bereits existierenden Objekt suchen. In [ShZ90] hingegen entfernt eine Funktion DupEliminate eventuelle Duplikate erst nach ihrer Erzeugung. Dieses Problem tritt allerdings nur bei partieller Instanziierung der Sichtklassen auf; werden alle Sichtklassen stets beim ersten Zugriff vollständig instanziiert, so kommt es zu keiner späteren Erzeugung von Sichtobjekten und die Tests können entfallen. Details zur Objektverwaltung folgen im Implementierungskapitel.

Eigenschaft 4
Die Objektidentität eines Sichtobjekts wird aus seinen Rückverweisen bestimmt.

Die bisherigen Eigenschaften beschreiben das Sichtsystem als verallgemeinert identitätserhaltend in Sinne von Abschnitt 2.2.3.3.

3.1.2 Granularität und Einbettung

a) Einordnung in den Typverband

Die Eigenschaften 1 und 2 fordern die Definition neuer Klassen zur Bereitstellung von Sichten. Da sowohl im Datenmodell von ODMG–93 als auch im Typsystem von C++ die Klassen in einem Verband angeordnet sind, stellt sich die Frage, wie man die neu entstehenden Klassen einordnet. Geht man davon aus, daß in vielen Fällen die Unterschiede zwischen einer Sichtklasse und der zugehörigen Originalklasse nicht allzu groß sind, so erscheint es naheliegend, die neue Klasse in unmittelbarer Nähe der Originalklasse in den Verband zu integrieren. Wird etwa eine Methode hinzugefügt, erhält man eine Subklasse, werden Attribute ausgeblendet, erhält man eine Superklasse. Während diese Vorgehensweise im objekterhaltenden Fall gut anwendbar ist, kann eine solche Einordnung im gewählten objektgenerierenden Modell zu Problemen führen. ODMG–93 definiert nämlich eine Inklusionsabhängigkeit im Klassenverband: Eine Instanz einer beliebigen Klasse ist gleichzeitig auch Instanz jeder Superklasse. Gliedert man nun die Sichtklasse oberhalb oder unterhalb der Originalklasse in den Verband ein und generiert neue Objekte, um sie zu füllen, so liegen in der jeweils allgemeineren Klasse sowohl die Originalobjekte als auch die Sichtobjekte gleichzeitig im Extent vor. Diese Vermischung führt jedoch zu Problemen, da nun verschiedene Instanzen einer Klasse gleichzeitig dasselbe Objekt darstellen, und man nicht mehr zwischen den verschiedenen Bedeutungen unterscheiden kann. Weiterhin kann es zu Rekursionen kommen, wenn etwa bei der Berechnung der Sicht die erzeugten Objekte automatisch in der Originalklasse auftauchen und hierfür wiederum Sichtobjekte generiert werden. Als Konsequenz ergibt sich, daß die neuen Klassen nicht als Sub- oder Superklassen in den bestehenden Verband eingeordnet werden dürfen. Alle Klassen müssen zwar gemäß ODMG–93 von PObject abgeleitet sein, außer dieser Klasse darf es aber keine Überschneidungen zwischen dem Datenbank- und dem Sichtverband geben.

Eigenschaft 5

Die in der Sicht definierten Klassen dürfen nicht in den Typverband der Datenbank eingebettet werden. Sie bilden einen unabhängigen Teilverband unter der Basisklasse PObject also ein eigenes Schema.

Dasselbe gilt, wenn von einer Sicht weitere Sichten abgeleitet werden: Jede neue Sicht muß ihren eigenen, unabhängigen Typverband verwenden. Innerhalb einer Sicht kann hingegen Vererbung verwendet werden.

b) Abgeschlossenheit

Eine Sicht muß in unserem Modell in sich abgeschlossen sein. Alle Relationships müssen zu anderen Sichtklassen führen und dürfen nicht auf originale Datenbankklassen zeigen. Einerseits bedeutet dies einen konsequenten Sichtentwurf, bei dem eine unkontrollierte Vermischung von Sicht- und Datenbankobjekten vermieden wird. Andererseits ist dies eine Folge der Bidirektionalität von Relationships in ODMG–93. Die Forderung, daß jede Relationship in beiden beteiligten Klassen definiert sein muß, verbietet Relationships zwischen Sichtklassen und Originalklassen, da eine Änderung der Originalklassen im Zuge einer Sichtdefinition unerwünscht, und im Sinne der Datenbankautonomie sogar verboten ist. Aus diesem Grunde ist es auch nicht möglich, die bereits erwähnten Rückverweise als Relationships zu implementieren. Sie müssen statt dessen als unidirektionale, objektwertige Attribute realisiert werden. Dasselbe gilt auch, wenn eine Sicht von einer anderen abgeleitet wird: Die Ausgangssicht ist hier ebenfalls als unveränderbar und abgeschlossen anzusehen.

Eigenschaft 6

Jede Sicht ist in sich abgeschlossen. Es gibt nur Relationships zwischen Klassen einer Sicht.

3.1.3 Herleitung und Spezifikation

Im Gegensatz zu relationalen Sichten können ganze Schemata nicht mehr mit einem reinen Anfrage-Ansatz definiert werden. Wie oben bereits erwähnt, enthält die Anfragesprache OQL keine Elemente zur Typdefinition. Typen müssen unabhängig von der Anfrage in der Datendefinitionssprache ODL spezifiziert werden, und Methoden werden ausschließlich in der Zielsprache programmiert. Solche Typänderungen sind in der normalen Anfrageumgebung folglich nicht durchführbar. Selbst rein strukturelle Änderungen, die mit Hilfe der Anfragesprache ausdrückbar sind, erfordern die Definition neuer Klassen, da die umformatierten Werte nicht an die Originalobjekte angehängt werden können. Die strukturelle Beschreibung einer Sicht erfolgt stets durch Definition einer Menge von Klassen mit Hilfe der Datendefinitionssprache.

Was noch fehlt ist das Verfahren, wie diese Klassen mit Objekten gefüllt werden. Genau wie im relationalen Fall geschieht dies durch Anfragen. Zuerst wird die Instanzmenge einer Sichtklasse mit Hilfe einer sogenannten *Extentklausel* definiert, die die Menge aller Sichtobjekte sowie deren Rückverweise festlegt. Im Gegensatz zu relationalen Sichten werden dabei die Datenfelder der Sichtobjekte noch nicht gefüllt. Wegen des üblicherweise navigatorischen Zugriffs auf Objekte, und wegen der Möglichkeit, Attributzugriffe im Objekt zu kapseln, ist es in meinem System sinnvoller, jedes Datenfeld mit einer eigenen Herleitungsklausel zu definieren. Bei einem Zugriff auf ein Datenfeld wird dann mit Hilfe der Herleitungsklausel des Feldes und der bei der

Sichtobjekterzeugung initialisierten Rückverweise der aktuelle Wert des Datenfelds bestimmt.[12]

Eigenschaft 7

Der Extent einer Sichtklasse wird über eine Anfrage (*Extentklausel*) definiert. Jedes Datenfeld eines Sichtobjekts wird ebenfalls über einen eigenen Anfrageausdruck bestimmt (*Attribut-* bzw. *Relationshipklausel*).

Die getrennte Herleitung von Instanzen und ihren Attribut- und Relationshipwerten führt dabei zu einer wesentlichen Vereinfachung von Spezifikation und Auswertung. Würde man die Objekte in einer einzigen Anfrage vollständig mit allen Datenfeldern instanziieren, so ergäbe sich ein sehr komplexer Anfrageausdruck. Außerdem müßten bei der Instanziierung auch alle referenzierten Klassen ebenfalls instanziiert werden, um die Relationships mit den Zielobjekten füllen zu können. Dies führt zum einen zu einer Kaskade von Objekterzeugungen entlang der transitiven Hülle der Relationships. Zum anderen bekommt man Probleme bei der Initialisierung der Inversen, da die Konstruktoren der referenzierten Klassen wiederum Konstruktoren der eigenen Klasse aufrufen, um die inversen Relationen mit Zielobjekten zu füllen. Eine saubere Auswertung dieser zyklischen Objekterzeugungen erfordert dabei einigen Aufwand, der durch die hier beschriebene Zweiphasigkeit sehr einfach umgangen wird: Bei der Extentbestimmung werden nur leere Objekte angelegt, Relationships werden also nicht verfolgt. Erst beim Zugriff auf eine Relationship müssen dann die Zielobjekte angelegt werden, die aber wiederum vorerst leer sind, so daß Zyklen nicht mehr auftreten. Eine Diskussion zu diesem Problem findet man im Anhang von [D4–2/1].

Methodenintegration

Ein wesentliches Element im objektorientierten Datenmodell ist das Vorhandensein von Methoden. Da meine Sichtklassen von der Struktur her normale ODMG-Klassen sind, kann man auch Methoden für sie definieren. Diese Methoden können völlig neu implementiert werden oder einfach geeignete Methoden der Originalklassen aufrufen.

Eigenschaft 8

Jede Sichtklasse darf neue Methoden definieren.

Durch die Definition von Sichten in Form von Klassen mit voller Methodenmächtigkeit unterstützt das hier vorgestellt Sichtsystem alle Transformationen aus Kapitel 2.2.3.2.

Zusammenfassung

Tabelle 5 faßt noch einmal die Einordnung des Systems gemäß Kapitel 2.2 zusammen:

Kategorie	Mein Ansatz	Eigenschaften
Art der Transformation	alle	1, 7, 8
Art des Ergebnisses	verallgemeinert identitätserhaltend	1, 2, 3, 4
Granularität	Schema	5, 6
Einbettung	nein	5, 6

Tabelle 5: Klassifikation meines Ansatzes

12. Diese Auswertung kann sowohl deklarativ/mengenorientiert, als auch prozedural/instanzorientiert erfolgen. Wegen der mangelnden Ausdrucksmächtigkeit von OQL in Bezug auf Updates werden Schreibzugriffe ausschließlich prozedural zur Verfügung gestellt (siehe Kapitel 4).

3.2 Das Sichtmodell: Virtuelle Klassen

Im folgenden wird für die acht Grundeigenschaften ein konkretes Sichtmodell entworfen. Die Eigenschaften legen fest, daß eine Sicht aus einer Reihe von Klassen besteht, die Struktur und Verhalten der Sichtobjekte definieren und mit Sichtobjekten instanziiert werden. Diese Klassen werden *virtuelle Klassen* genannt, da ihre Objekte nicht zur Datenbank gehören, sondern aus anderen Datenbankobjekten berechnet werden. Der Extent einer virtuellen Klasse wird durch Auswertung einer Anfrage auf den zugehörigen Originalklassen bestimmt. Die Objekte und Attributwerte sind also *abgeleitet* (*derived*). Neben den berechneten Attributen kann eine virtuelle Klasse aber auch normale Attribute enthalten. Diese können während der Arbeit mit den Sichtobjekten benutzt werden, haben aber keinen Einfluß auf die Originalobjekte. Das heißt, daß Änderungen an diesen Attributen nicht in die Datenbank geschrieben werden und spätestens mit dem Deaktivieren des Sichtobjekts verloren sind.

Definition 3.1: Virtuelle Klasse, virtuelles Objekt
Eine *virtuelle Klasse* ist eine Klasse, deren Extent berechnet wird und nicht persistent vorliegt. Die Instanzen der Klasse heißen *virtuelle Objekte* und werden zur Laufzeit mittels einer Anfrage aus anderen Datenbankklassen erzeugt. Ihre Attributwerte werden aus den Attributen der Originalobjekte berechnet, und Updates werden an diese Originalobjekte weitergeleitet (Identitätserhaltung).

Definition 3.2: Virtuelles Datenfeld, transientes Datenfeld (resp. Attribut/Relationship)
Ein Datenfeld, das aus Datenfeldern und Methoden der Originalobjekte berechnet wird, heißt *virtuelles* oder *abgeleitetes Datenfeld*. Datenfelder, die keinen Bezug zu den Originalobjekten haben, heißen *transiente Datenfelder.* Die Bezeichnungen *virtuell* und *transient* gelten entsprechend für Attribute und Relationships.

Die Berechnung des Extents und der einzelnen Datenfelder wird durch spezielle OQL-Klauseln beschrieben. Wie oben bereits beschrieben, muß bei der Objektgenerierung die Verwaltung eines Rückverweises beachtet werden. Hierfür dient ein ausgezeichnetes Attribut namens orig (von origin, Ursprung).

Definition 3.3: Extentklausel, Basisklasse
Jede virtuelle Klasse wird durch Auswertung einer Anfrage auf den *Basisklassen* der virtuellen Klasse instanziiert. Diese Anfrage wird *Extentklausel (extent derivation clause)* genannt. In der Anfrage wird ausgehend von den Extents der Basisklassen der Konstruktor der virtuellen Klasse für jede gewünschte Objektkombination aufgerufen.

Definition 3.4: Basisobjekt, orig-Attribut
Jedes virtuelle Objekt verwaltet einen Rückverweis auf seine *Basisobjekte*, also auf jene Instanzen der Basisklassen, von denen das Objekt abgeleitet worden ist. Dieser Verweis wird in einem privaten Attribut namens orig abgelegt.

Definition 3.5: Attributklausel
Die Attributwerte eines virtuellen Objekts werden aus den Attributwerten der Basisobjekte berechnet. Für jedes virtuelle Attribut ist hierzu eine OQL-Anfrage angegeben, die *Attributklausel (attribute derivation clause).*

Definition 3.6: Relationshipklausel
Die Relationships eines virtuellen Objekts werden ebenfalls aus den Relationshipwerten der Basisobjekte berechnet. Die zugehörige OQL-Anfrage heißt in diesem Fall *Relationshipklausel (relationship derivation clause).*

Zusätzlich zur Definition virtueller Datenfelder können für eine Sichtklasse auch Methoden definiert werden; für deren Definition ist allerdings keine deklarative Ableitungsklausel vorgesehen. Dies ist begründet in der Annahme, daß in den meisten Fällen echter Methodencode notwendig sein wird, um die Herleitung ausdrücken zu können. Selbst bei reinen Parameteranpassungen stieße ein reiner OQL-Ausdruck zu schnell an seine Grenzen. Mit dieser Entscheidung folge ich auch dem Standard, der ebenfalls in der Schemadefinition nur Methodensignaturen zuläßt. Die eigentliche Methodenfunktionalität wird direkt in der jeweiligen Sprachbindung hinzugefügt.

Da OQL keine Sprachelemente zur Definition von Updates enthält, und da das objektorientierte Modell im allgemeinen zu kompliziert ist, um Inversen automatisch aus den Herleitungsklauseln zu bestimmen, werden Updates ausschließlich in Form von Methoden zur Verfügung gestellt. In der Sprachbindung wird jedes virtuelle Datenfeld als Paar von Zugriffsmethoden realisiert. Die Lesemethode wird hierbei aus der Ableitungsklausel generiert, die Schreibmethode muß explizit vom Entwickler bereitgestellt werden (vgl. [HeZ90] und Ausblick). Werden beide Zugriffsmethoden vom Entwickler beigesteuert, können auch Ableitungen modelliert werden, deren Darstellung als OQL-Ausdruck zu kompliziert wäre. (Zum Vergleich von deklarativem und prozeduralem Zugriff siehe Abschnitt 3.5.)

3.3 Verfeinerung des Modells: Konvertierung und Import

Das eben beschriebene Modell erlaubt, neue Klassen zu definieren, die aus anderen Klassen hergeleitet werden. Betrachtet man die Architektur in Bild 6 (Seite 33), so kann man speziell bei der Datenbankintegration diese virtuellen Klassen funktional in Untergruppen einteilen. Jene virtuellen Klassen, die dazu dienen, externe Objekte verfügbar zu machen, nennt man *Importklassen*; bei ihnen zeigen die Rückverweise auf Objekte in einer anderen Datenbank. Die restlichen virtuellen Klassen, die der Homogenisierung und Vereinigung der Daten dienen, werden *Konvertierungsklassen* genannt. Ihnen ist gemeinsam, daß die Rückverweise auf eines oder mehrere Objekte in der gleichen Datenbank verweisen.

Eine Sicht kann entweder an eine Datenbank gebunden sein oder als sogenannte *„Middleware“* ohne eigene Persistenzschicht laufen. Der erste Fall tritt ein, wenn reine Sichtfunktionalität ohne Integrationsanteile benötigt wird, oder wenn eine der beteiligten Datenbanken der Sicht zur persistenten Ablage ihrer Daten zur Verfügung steht (*home database*). In diesem Fall zählen sowohl die persistenten Originalobjekte dieser Datenbank als auch die Sichtinstanzen zu den Objekten der Datenbank. Im Falle einer Middleware ohne Persistenz bezeichne ich der Einfachheit halber das Sichtsystem ebenfalls als „Datenbank“, obwohl es nur transiente Objekte enthält und alle Zugriffe auf persistente Objekte über Importklassen erfolgen.

Definition 3.7: Klassenkategorie

Die Begriffe *normal (=nicht-virtuell)*, *importierend* und *konvertierend* definieren drei *Klassenkategorien*.

3.3.1 Konvertierungsklassen

Definition 3.8: Konvertierungsklasse

Eine *Konvertierungsklasse* ist eine virtuelle Klasse, deren orig-Referenzen auf Objekte verweisen, die in der gleichen Datenbank wie die Konvertierungsklasse liegen.

Da sich Sichtobjekte und Basisobjekte im gleichen Objektraum befinden, kann man davon ausgehen, daß vom Sichtobjekt aus alle Basisobjekte direkt und effizient zugreifbar sind. Folglich können die Herleitungsoperationen direkt auf den Basisobjekten ausgeführt werden. Ein Zugriff auf ein Datenfeld oder eine Methode eines Sichtobjekts wird also direkt in einen entsprechenden Zugriff auf die Basisobjekte übertragen. Virtuelle Datenfelder werden nicht im Sichtobjekt materialisiert sondern jedesmal neu ausgewertet. Nur die transienten Datenfelder und das orig-Attribut werden im Objekt materialisiert. Das orig-Attribut ist stets eine Struktur mit mindestens einer Objektreferenz. Bei der Vereinigung von Klassen können auch mehrere Referenzen auftreten, und zur Identifikation von Objektteilen können noch weitere literale Werte enthalten sein. Zu den Konvertierungsklassen zählen sowohl die Homogenisierungsklassen als auch die Vereinigungsklassen aus Kapitel 2.2, da in beiden Fällen die durch orig adressierten Objekte im gleichen Objektraum liegen.

3.3.2 Importklassen

Definition 3.9: Importklasse

Eine *Importklasse* ist eine virtuelle Klasse, deren orig-Referenz auf eine externe Klasse verweist.

Wie bereits in Kapitel 2.3.2 angesprochen, unterscheiden sich Importklassen in einer Reihe von Eigenschaften von Konvertierungsklassen:

- Symbolischer Rückverweis

  Da sich das Basisobjekt einer Import-Instanz in einer anderen Datenbank befindet, kann man nicht direkt mit Hilfe einer Objektreferenz darauf zugreifen, sondern muß über ein Netzprotokoll gehen.[13] Die hierfür notwendige globale Objekt-Id (GOID) wird als Rückverweis im orig-Attribut gespeichert (siehe Seite 89 und Anhang H.7).

- Reine 1-zu-1-Abbildung

  Der Einfachheit halber sind in Importklassen keine strukturellen Änderungen erlaubt. Sie dienen ausschließlich dem Zweck, externe Klassen sichtbar zu machen. Grundsätzlich existiert für jedes externe Objekt genau ein Sichtobjekt, dessen Datenfelder und Methoden in Name, Typ und Signatur mit den Datenfeldern und Methoden des Originalobjekts übereinstimmen. Um Konflikte und unnötige Datentransfers zu vermeiden, gelten aber folgende Ausnahmen:

  – Der Extent der Sichtklasse kann durch eine Selektion auf eine Teilmenge der Originalobjekte eingeschränkt werden.

  – Einzelne Datenfelder und Methoden können in der Importklasse ausgelassen werden, um das Interface auf die notwendigen Elemente zu beschränken.

  – Die Importklassen können gegebenenfalls umbenannt werden, wenn es sonst zu Namenskonflikten käme. Ein gemeinsamer Präfix für alle Klassen einer Datenbank ist hier der einfachste Ansatz.

  Durch die strikte 1-zu-1-Abbildung kann es keine Relationships zwischen Importklassen und Konvertierungsklassen oder normalen Klassen geben.

---

13. Direkte externe Objektreferenzen, wie sie etwa von CORBA oder Java-RMI angeboten werden, sind in ODMG–93 nicht vorgesehen, könnten aber durch geeignete Modellierung des Ref-Datentyps in der Sprachbindung realisiert werden. Diesen Ansatz habe ich nicht verfolgt, da in meinem System die Cache-Funktionalität der Importklassen ausdrücklich erwünscht ist.

- Reduzierte Spezifikation

  Die Funktionalität jedes Datenfelds und jeder Methode ist eindeutig als Zugriff auf das entsprechende Element des externen Objekts definiert. Die Definition konkreter Herleitungsklauseln kann somit entfallen und wird durch einen Ausdruck ersetzt, der angibt, von welcher externen Klasse die Objekte importiert werden sollen. Die Herleitung der einzelnen Datenfelder und der Methodencode werden dann automatisch generiert.

- Zwischenspeicherung (Caching)

  Da die Basisobjekte der Importinstanzen in einer anderen Datenbank liegen, würde der direkte Zugriff auf das externe Objekt bei jedem Zugriff auf ein Sichtattribut zu einer extrem hohen Netzlast führen und somit ineffizient werden. Aus diesem Grunde werden die Attribute des Basisobjekts im Sichtobjekt gespeichert, um unnötige Netzzugriffe zu vermeiden. Details zum Caching-Verfahren folgen in Abschnitt 4.2.5.2 und in Kapitel 5.

Definition 3.10: Cache-Datenfeld
Ein *Cache-Datenfeld* ist ein virtuelles Datenfeld einer Importklasse. Sein Wert wird im Importobjekt gespeichert. Ein Cache-Datenfeld ist entweder ein *Cache-Attribut* oder eine *Cache-Relationship*.

3.3.3 Vererbungsbeziehungen

Die Vererbung ist ein wesentliches Element objektorientierter Programmierung und wird daher auch im Sichtsystem unterstützt. Vererbungsbeziehungen zwischen Sichtklassen und Originalklassen können dabei nicht auftreten, da jede Sicht ihren eigenen, unabhängigen Typverband erhält. Auch zwischen Konvertierungsklassen und Importklassen einer Sicht ist eine Vererbungsbeziehung nicht möglich: durch die Inklusionsabhängigkeit käme es sonst zur Vermischung von importierten und konvertierten Objekten in einer Klasse. Genauso können normale Klassen und virtuelle Klassen nicht voneinander erben, da sich die expliziten und automatischen Instanziierungsmechanismen widersprechen. Eine Sicht besteht folglich aus drei unabhängigen Typhierarchien für die normalen, importierenden und konvertierenden Klassen. Während normale Klassen nur den Vererbungsrichtlinien von ODMG–93 folgen müssen, sind bei virtuellen Klassen bestimmte Konsistenzbedingungen zu erfüllen.

Bei Importklassen muß die Vererbungshierarchie mit der entsprechenden Vererbungshierarchie der zugehörigen Basisklassen übereinstimmen.

Konsistenzbedingung zur Vererbung von Importklassen:
Eine Importklasse I_2 ist genau dann Subklasse einer Importklasse I_1, wenn die Basisklasse B_2 von I_2 Subklasse der Basisklasse B_1 von I_1 ist.

Diese Forderung ergibt sich direkt aus Inklusionsabhängigkeit und Objektidentität: da eine Instanz von I_2 automatisch auch Instanz von I_1 ist, muß ein entsprechendes Basisobjekt in B_1 existieren (und umgekehrt).

Bei Konvertierungsklassen ist die Situation komplizierter, da neben einer semantisch korrekten Vererbungshierarchie auch die unterschiedlichen Typen der orig-Attribute beachtet werden müssen. Die ursprüngliche Idee war, in den verschiedenen Konvertierungsklassen einer Vererbungshierarchie unterschiedliche orig-Attribute zuzulassen, sofern sich diese nicht widersprechen, sie also sowohl strukturell als auch inhaltlich kompatibel sind. Dieser Ansatz führt jedoch zu einer technisch sehr umständlichen Lösung, da man neben der Vererbung der Konvertierungsklassen

noch eine parallele Vererbung der orig-Attributtypen durchführen muß. Die Zuweisung und Konstruktion solcher orig-Werte erfordert zusätzliche Indirektionen, Zugriffsfunktionen, Konstruktoren und so weiter, was in einem sehr hohen Aufwand für die interne Typverwaltung resultiert.

Betrachtet man jedoch die Inklusionsabhängigkeit, so folgt, daß jede Instanz einer Konvertierungsklasse auch Instanz der allgemeinsten Konvertierungsklasse sein muß, von der diese Klasse erbt. Bereits in dieser allgemeinen Klasse muß die Objektidentität also eindeutig bestimmbar sein, um die Instanz ansprechen und von anderen Instanzen unterscheiden zu können. Zusätzliche orig-Information in Subklassen kann folglich nicht zur Objektidentität beitragen, sondern kann durch Ausnutzung zusätzlicher Detailinformation höchstens zur Klassfikation herangezogen werden. Hieraus kann man die folgende Konsistenzbedingung für einen vereinfachten Ansatz ableiten:

Konsistenzbedingung zur Vererbung von Konvertierungsklassen:

Eine Konvertierungsklasse, die nicht Subklasse einer anderen Konvertierungsklasse ist, definiert ein orig-Attribut, das der Zuordnung zu den Basisobjekten dient. Alle Subklassen müssen dieses Attribut verwenden; eine Neudefinition ist nicht erlaubt.

3.3.4 Methoden von Importklassen

Neben der Vererbungshierarchie müssen auch Methoden als wesentliches Element des Objektmodells in der Modellierung berücksichtigt werden. Die Methoden normaler Klassen haben keine sichtspezifische Besonderheit und werden gemäß ODMG–93 direkt in der jeweiligen Sprachbindung ausprogrammiert. Dasselbe gilt für Konvertierungsklassen, da auf Grund der potentiellen Komplexität möglicher Methodenherleitungen eine deklarative Spezifikation nicht vorgesehen ist. Die überwiegende Mehrheit dieser Konvertierungsmethoden wird jedoch keine eigene Funktionalität anbieten, sondern einfach eine Hülle für den Aufruf geeigneter Methoden der Originalobjekte sein. Bei der Untersuchung von Importklassen muß man die Randbedingungen des externen Zugriffs beachten. Für die Austauschbarkeit und Kompatibilität von ausführbarem Code gibt es die folgenden drei grundsätzlichen Möglichkeiten:

- *Dynamischer Codeaustausch*

  Der dynamische Codeaustausch ist ein Idealmodell objektorientierter Modellierung. Hier geht man davon aus, daß das Objekt eine in sich abgeschlossene Einheit darstellt, die zwischen verschiedenen Rechnern ausgetauscht werden kann. Eine Datenbankanwendung fordert zum Beispiel ein bestimmtes Objekt aus der Datenbank an und bekommt dieses Objekt mit all seinen Datenfeldern und Methoden zugestellt. Sie kann dann die Daten bearbeiten, die Methoden direkt ausführen, und gegebenenfalls das modifizierte Objekt in die Datenbank zurückschreiben. Ein solches Modell setzt jedoch die Ausführbarkeit der Methodenimplementierung in allen Anwendungssystemen voraus. Dies ist zum Beispiel dann möglich, wenn die Datenbank zur Methodenimplementierung eine eigene Manipulationssprache verwendet, die in der Datenbank gespeichert, zur Anwendung übertragen und dann dort ausgeführt werden kann (*stored procedures*). Ein anderer Bereich, in dem dynamischer Codeaustausch bereits Verwendung findet, sind die sogenannten *Applets* im World Wide Web, deren Code automatisch zum Anwender transportiert wird und in dessen Browser abläuft.

- *Statischer Codeaustausch*

  Ist eine solche generische Methodenimplementierung nicht verfügbar, kann statt dessen ein statischer Codeaustausch erfolgen. Hierbei werden die Methodenimplementierungen in einer Bibliothek gesammelt, die sowohl zur Datenbank als auch zu den Anwendungen

hinzugebunden werden muß. Für die Objekte steht so auf beiden Seiten der Verbindung eine Umgebung zur Verfügung, in der die Methoden direkt ausführbar sind.

- *Kein Codeaustausch*

  Die letzte Alternative schließt jeglichen Codetransfer aus. In einer heterogenen Umgebung muß man davon ausgehen, daß die Methodenimplementierungen an eine bestimmte Ausführungsumgebung gebunden sind und weder dynamisch noch statisch übertragen werden können. In einem solchen Fall können zwar die Daten zwischen der Datenbank und den Anwendungen ausgetauscht werden, die Methoden selbst müssen jedoch stets innerhalb der Datenbank ausgeführt werden.

In dem hier vorgestellten System gehe ich von einer heterogenen Umgebung aus, in der kein Codeaustausch möglich ist. Die Methoden der Importklassen sind folglich als Hüllen zu implementieren, die die Parameter an die Datenbank übertragen, dort die Ausführung der Originalmethode veranlassen und dann das Ergebnis zurückliefern. Auf die dabei notwendigen Maßnahmen zur Konsistenzerhaltung wird in Kapitel 5 eingegangen.

3.4 Sichtspezifikation

Nach der vorangegangenen Modelldefinition folgt nun die eigentliche Sichtspezifikation. Da es in ODMG–93 bisher noch keine Syntax für Sichten gibt, muß die Schemadefinitionssprache ODL geeignet erweitert werden, um virtuelle Klassen als solche zu kennzeichnen und ihre Herleitung zu spezifizieren. Da man bei gegebener ODMG-Datenbank aber eventuell auf die Verwendung des originalen ODL-Präprozessors angewiesen ist, der diese Erweiterungen nicht kennt, stelle ich zusätzlich eine Spezifikationsvariante zur Verfügung, die alle sichtspezifischen Teile in eine separate Datei auslagert.

3.4.1 Spezifikationsansätze

Die meisten objektorientierten Datenbanksysteme betten Datenbankobjekte wie normale Objekte in die Programmiersprache ein und kombinieren so klassische Datenbankkonzepte und Programmierkonzepte. Bei der Sichtspezifikation kann man nun diesen beiden grundsätzlichen Linien folgen. Aus der Sicht klassischer Datenbanken ist eine Sicht eine deklarative Anfrage, die immer dann ausgewertet wird, wenn auf die Sichtinstanzen zugegriffen wird. Das Ergebnis einer solchen *definierenden Anfrage* stellt hierbei die Instanzmenge der Sicht dar. Die Vorteile eines solchen Verfahrens sind die leichte Spezifizierbarkeit und die Möglichkeit, die Sichtinstanziierung abhängig von der Benutzeranfrage zu optimieren. Der Nachteil ist, daß eine solche Sicht nur von einem Anfragesystem ausgewertet werden kann. Der andere Ansatz zur Sichtspezifikation in objektorientierten Systemen folgt den Programmiersprachen. Da jede Datenbankklasse in der Programmiersprache repräsentiert ist, kann man auch die Sicht komplett in der Programmiersprache ausprogrammieren. Sichtklassen werden also zum Beispiel durch eine Klassenmethode instanziiert, und alle Zugriffe auf die Sichtobjekte führen Objektmethoden aus, die die Berechnung der jeweiligen Werte durchführen. Hierbei hat man die volle Mächtigkeit der Programmiersprache zur Verfügung und kann beliebig komplexe Sichtberechnungen durchführen. Außerdem können Anwendungsprogramme direkt auf die Objekte zugreifen, ohne über ein Anfragesystem gehen zu müssen. Die wesentlichen Nachteile sind der umfangreiche Programmieraufwand und der Instanziierungszwang, da auf Sichtobjekte nur zugegriffen werden kann, wenn sie als Objekte instanziiert vorliegen.

In meinem Sichtsystem kombiniere ich beide Ansätze: Eine deklarative Spezifikationssprache gibt eine einfache Möglichkeit, Sichtherleitungen zu definieren, und kann vom Anfragesystem

zur Optimierung ausgenutzt werden. Kompliziertere Herleitungen können dabei in Methoden verborgen werden. Aus dieser Spezifikation wird eine prozedurale Darstellung generiert, die zusätzlich ein direktes Interface für Anwendungsprogramme bereitstellt (vgl. Abschnitt 3.5). Im folgenden wird die deklarative Sichtspezifikation beschrieben. Die Eigenschaften der prozeduralen Darstellung und ihre Generierung werden im Implementierungskapitel behandelt.

3.4.2 Eingebettete Spezifikation

Für die deklarative Spezifikation verwende ich soweit möglich die vorhandenen Sprachelemente von ODMG–93. Die Anfragesprache OQL allein reicht für eine vollständige Sichtdefinition nicht aus. Während man mit OQL zwar Daten umstrukturieren und auch Objekte erzeugen kann, ist es hingegen nicht möglich, neue Klassen zu erzeugen. Alle generierten Objekte müssen zu bereits existierenden Klassen gehören. Da eine Sicht, um Umstrukturierungen zuzulassen, aus neuen Klassen besteht, muß man für die Definition dieser Klassen auf die Datendefinitionssprache ODL zurückgreifen. Die Grundidee zur Spezifikation von Sichten in ODMG–93 ist folglich, ODL um minimale sichtspezifische Elemente zu erweitern und mit OQL zu kombinieren. Die erweiterte ODL-Syntax nenne ich ODL*. Entsprechend dem oben beschriebenen Ansatz muß eine ODL-Klassendefinition um die folgenden Elemente erweitert werden:

- Virtuelle Klassen müssen als solche gekennzeichnet werden.
- Jede virtuelle Klasse muß eine Extentklausel erhalten.
- Für jedes virtuelle Attribut und jede virtuelle Relationship muß eine Ableitungsklausel angegeben werden.
- Jede virtuelle Klasse muß ein privates orig-Attribut erhalten.

Alle Ableitungsklauseln können als einfache OQL-Anfragen geschrieben werden. An den entsprechenden Stellen in der Klassendefinition wird hierzu der Term query aus der OQL-Grammatik in Anhang E verwendet. Zusätzlich kann jedes Attribut und jede Relationship noch explizit als **virtual** gekennzeichnet werden. Dies ist aber optional, da die Eigenschaft automatisch aus dem Vorhandensein einer Ableitungsklausel folgt.

Jede Sichtklasse besitzt ein vordefiniertes Attribut namens orig. Dieser Name ist reserviert und darf nicht für Benutzerdaten verwendet werden. Dieses Attribut sollte vor dem Benutzer versteckt werden, da es nur internen Zwecken dient und nicht zum funktionalen Interface der Klasse gehört. Da es in ODMG–93 keine privaten Attribute gibt, sollte das Schlüsselwort **private** eingeführt werden. Dieses ist jedoch für die Funktionsfähigkeit des Sichtsystems nicht unbedingt erforderlich. Bei der nun folgenden Syntaxbeschreibung muß zwischen Konvertierungsklassen und Importklassen unterschieden werden.

3.4.2.1 Eingebettete Spezifikation von Konvertierungsklassen

Die Spezifikation einer Konvertierungsklasse enthält alle der gerade beschriebenen Elemente. Die verwendete Syntax ist schematisch im folgenden Beispiel gegeben; eine vollständige Grammatik findet sich in Anhang D.

Beispiel 3.1:

Die Spezifikation einer Konvertierungsklasse in erweitertem ODL (ODL*) sieht folgendermaßen aus; die Erweiterungen sind unterstrichen:

```
interface mytype : supertype_list
    // type properties
    ( extent extentname is query;
      keys property_list ) : virtual
    {
    // instance properties
    private   attribute     OrigType     orig;
    [ virtual ] attribute   aType        attname is query;
    ...
    [ virtual ] relationship OtherType   relname inverse OtherType::invrel is query;
    ...
    // instance methods
    long        compute_value ( in long increment ) raises ( ExOverflow );
    ...
    };
```

Der Typ OrigType ist nicht vordefiniert sondern hängt von der Definition der jeweiligen Klasse ab. Er stellt entweder eine Referenz auf eine einzelne Basisklasse dar oder eine Struktur mit Elementen, die auf mehrere Basisklassen verweisen. Wenn für die Identifikation der Objekte noch weitere Daten, wie etwa Schlüsselwerte, notwendig sind, so werden sie ebenfalls in dieser Struktur abgelegt. Obwohl orig Objektreferenzen enthält, wird es nicht als Relationship implementiert, da die notwendige Inverse Änderungen an den Originalobjekten erfordern würde und außerdem strukturierte Relationships nicht erlaubt sind. ■

Die angegebene Syntax erfüllt dabei die folgenden Aufgaben:

- Sie definiert ein Schema für die Sicht.

  Mit dem Grundgerüst aus ODL wird die Struktur jeder einzelnen Sichtklasse festgelegt. Alle Konvertierungsklassen werden hierbei von einer gemeinsamen Wurzelklasse DerivedClass abgeleitet, die eine Grundfunktionalität für die Implementierung zur Verfügung stellt. Durch das neue Schlüsselwort **virtual**, das die vorhandenen Klassentypen **persistent** und **transient** ergänzt, wird die Klasse eindeutig als Sichtklasse gekennzeichnet.

- Sie definiert einen Rückverweis auf die Originalobjekte.

  Um sowohl Herleitungen als auch Updates ausdrücken zu können, muß jedes Objekt eine Referenz auf die zugrunde liegenden Originalobjekte aufweisen. Das private orig-Attribut dient diesem Zweck. Es wird bei der Erzeugung des Sichtobjekts mit den zugehörigen Werten belegt und gehört nicht zum öffentlichen Interface der Klasse. Innerhalb der Sichtdefinitionen ist das Attribut aber für alle Klassen sichtbar, um auch Herleitungen, die mehrere Klassen betreffen, ausdrücken zu können; dies ist zum Beispiel für die Berechnung von Relationships notwendig. Um die Konsistenzbedingung von Seite 48 zu erfüllen, dürfen nur die direkt von DerivedClass abgeleiteten Konvertierungsklassen eine orig-Definition enthalten. Alle anderen Konvertierungsklassen müssen das geerbte orig-Attribut verwenden. Benötigt eine Subklasse zusätzliche Informationen über die Basisobjekte, müssen diese entweder navigatorisch gewonnen werden, oder bereits in der orig-Definition der Superklasse enthalten sein, auch wenn sie dort noch nicht benötigt werden.

- Sie definiert einen Ausdruck zur Berechnung der Instanzmenge (Extent)

  Die Extentklausel ist eine OQL-Anfrage, die eine Menge von Instanzen der Sichtklasse liefert. Die Anfrage führt die notwendigen Selektions-, Projektions- und Join-Operationen auf den zugrunde liegenden Basisklassen aus und ruft für jedes identifizierte Sichtobjekt einmal den Konstruktor der Sichtklasse auf. Durch Ausführung dieser Anfrage wird die

Sichtklasse vollständig instanziiert. Die Extentklausel wird mit dem neuen Schlüsselwort **is** direkt hinter dem Extent-Namen eingefügt. Die Syntax ähnelt somit der Definition benannter Anfragen in OQL (**define** identifier **as** query;). Da nur die jeweils oberste Klasse einer Vererbungshierarchie ein orig-Attribut definiert, werden auch nur in der zugehörigen Extentklausel neue Objekte erzeugt. Die Extentklauseln der von dieser Klasse abgeleiteten Konvertierungsklassen verwenden statt dessen eine Selektion auf den Extents ihrer direkten Superklassen, deren Ergebnismenge mit Hilfe eines Cast-Operators auf den spezifischeren Typ gebracht wird. Zur Laufzeit wird dann durch Analyse der Selektionsprädikate entschieden, in welcher Subklasse das Objekt zu erzeugen ist.[14]

- Sie definiert Ausdrücke zur Berechnung virtueller Attribute.

  Genau wie beim Extent wird auch die Berechnung von virtuellen Attributen mit Hilfe eines OQL-Ausdrucks vorgenommen. Für jedes virtuelle Attribut wird die zugehörige Attributklausel mittels **is** query an die Attributdeklaration angehängt. Im Gegensatz zur Extentberechnung bezieht sich ein Attributwert aber immer auf das aktuelle Sichtobjekt, das durch das Schlüsselwort this zugreifbar ist.[15] Üblicherweise greift eine Attributklausel über das orig-Attribut von this auf die dem Sichtobjekt zugrunde liegenden Originalobjekte zu und berechnet so den Attributwert des Sichtobjekts. Die Existenz der Ableitungsklausel identifiziert das Attribut eindeutig als virtuell. Zu Gunsten der Lesbarkeit kann diese Eigenschaft durch Hinzufügen von **virtual** vor der Attributdeklaration ausdrücklich hervorgehoben werden.

- Sie definiert Ausdrücke zur Berechnung virtueller Relationships.

  Relationships werden genau wie Attribute über eine angehängte OQL-Anfrage berechnet. Auf Grund der Abgeschlossenheit der Sicht kann aber das Ergebnis nicht direkt aus den Objektreferenzen der Originalobjekte hergeleitet werden. Zu jedem referenzierten Objekt muß statt dessen das zugehörige Sichtobjekt gefunden und verwendet werden. Eine virtuelle Relationship wird deshalb in den meisten Fällen in Form einer Selektion über die referenzierte Sichtklasse dargestellt werden, bei der die OIDs oder Schlüsselwerte der referenzierten Originalobjekte als Selektionskriterium dienen.

- Sie erlaubt die Definition transienter Attribute und Relationships.

  Jedes Datenfeld, das nicht durch eine Herleitungsklausel spezifiziert ist, wird als transientes Datenfeld des Sichtobjekts aufgefaßt. Das Datenfeld wird bei der Objekterzeugung mit einem Standardwert initialisiert und kann durch den Benutzer bzw. die Anwendung mit neuen Daten belegt werden. Die entsprechenden Werte werden aber nur transient im Hauptspeicher gehalten. Da keine Abbildung auf die persistenten Datenbankobjekte vorliegt, können die Werte nicht in der Datenbank abgelegt werden und gehen somit beim Deaktivieren oder Löschen des Sichtobjekts verloren.

- Sie erlaubt die Definition von Methoden.

  Genau wie bei normalen ODMG-Klassen können Methodensignaturen angegeben werden. Zum einen dienen diese Methoden dazu, Methoden der Originalobjekte aufzurufen. Zum anderen kann durch sie neues Verhalten und zusätzliche Sichtfunktionalität realisiert

14. Alternativ könnte man alle Instanzen direkt in den Subklassen erzeugen und die Extentklausel der Superklasse durch Vereinigung der Extentklauseln aller Subklassen, erweitert um einen Ausdruck zur Erzeugung der direkten Instanzen, bilden. Das verwendete Verfahren verspricht aber einfachere Ausdrücke und einen leichteren Zugehörigkeitstest.

15. Das Schlüsselwort this erscheint zwar nicht in der OQL-Grammatik (Anhang E), gehört aber dennoch zum ODMG–93-Standard. Es wird dort zur Auswertung von Collection-Queries verwendet (siehe [Cat94, S. 129]).

werden, die mit Hilfe von reinem OQL nicht ausdrückbar sind. Die Methoden werden im Rahmen der Sprachbindung mit dem eigentlichen Code versehen. Eine deklarative Spezifikation zur Ableitung von Methoden ist nicht vorgesehen. Genau wie bei den Attribut- und Relationshipklauseln kann innerhalb der Methodenimplementierungen mittels des orig-Attributs auf die Datenfelder und Methoden der Basisobjekte zugegriffen werden.

Die Schemainformation und die Ableitungsklauseln werden im Data Dictionary der Sichtdatenbank abgelegt. Wie bei relationalen Sichtdefinitionen können die Herleitungsklauseln von der Anfrageauswertung ausgelesen und in die Benutzeranfrage eingefügt werden. Eine Beschreibung der Anfrageauswertung und der Möglichkeiten zur Optimierung wird im Implementierungskapitel gegeben.

Beispiel 3.2:

Der folgende Code gibt ein einfaches Beispiel für die Spezifikation einer Konvertierungsklasse.[16]

```
typedef struct { ADRDB_Person person; } Person_OrigType;

interface Person : DerivedClass
    (   extent persons is
                    select Person( orig: struct( person:p ) )
                    from p in adrdb_persons;
        key fullname ) : virtual
    {
    private    attribute      Person_OrigType  orig;
               attribute      string           fullname     is this.orig.person.fname + " "
                                                            + this.orig.person.lname;
               attribute      string           birthdate    is this.orig.person.birthdate;
               relationship   Address          address      inverse Address::residents is
                   element (
                       select a
                       from a in addresses
                       where a.orig.address = this.orig.person.address);
    };
```

Der Rückverweis, also das orig-Attribut, zeigt auf eine einzelne Basisklasse namens ADRDB_Person. Wir können hier annehmen, daß es sich um eine Importklasse handelt, die Personendaten aus einer Adreßdatenbank importiert. Die Extentklausel erzeugt für jede Instanz p dieser Basisklasse eine neue Instanz der Sichtklasse und trägt p als Rückverweis in das jeweilige orig-Attribut ein. Die Attributklauseln verwenden dann orig, um direkt auf die Attributwerte von p zuzugreifen: Zum Beispiel wird bei einem Zugriff auf Person::fullname über orig zum zugehörigen ADRDB_Person-Objekt navigiert, dessen fname- und lname-Attribute ausgelesen und die Kombination der gefundenen Werte an den Aufrufer zurückgeliefert. Die Relationship zur Sichtklasse Address wird hingegen durch Selektion berechnet. Beim Zugriff auf diese Relationship wird genau jene Instanz von Address ausgewählt, die von dem durch p referenzierten Basis-Adreßobjekt (ADRDB_Person::address) abgeleitet worden ist. Bild 7 veranschaulicht diese Beziehungen graphisch. ■

16. Im Gegensatz zum Beispiel in Anhang C verzichten wir auf einen Join mit der Autorendatenbank, strukturieren dafür aber den Namen neu.

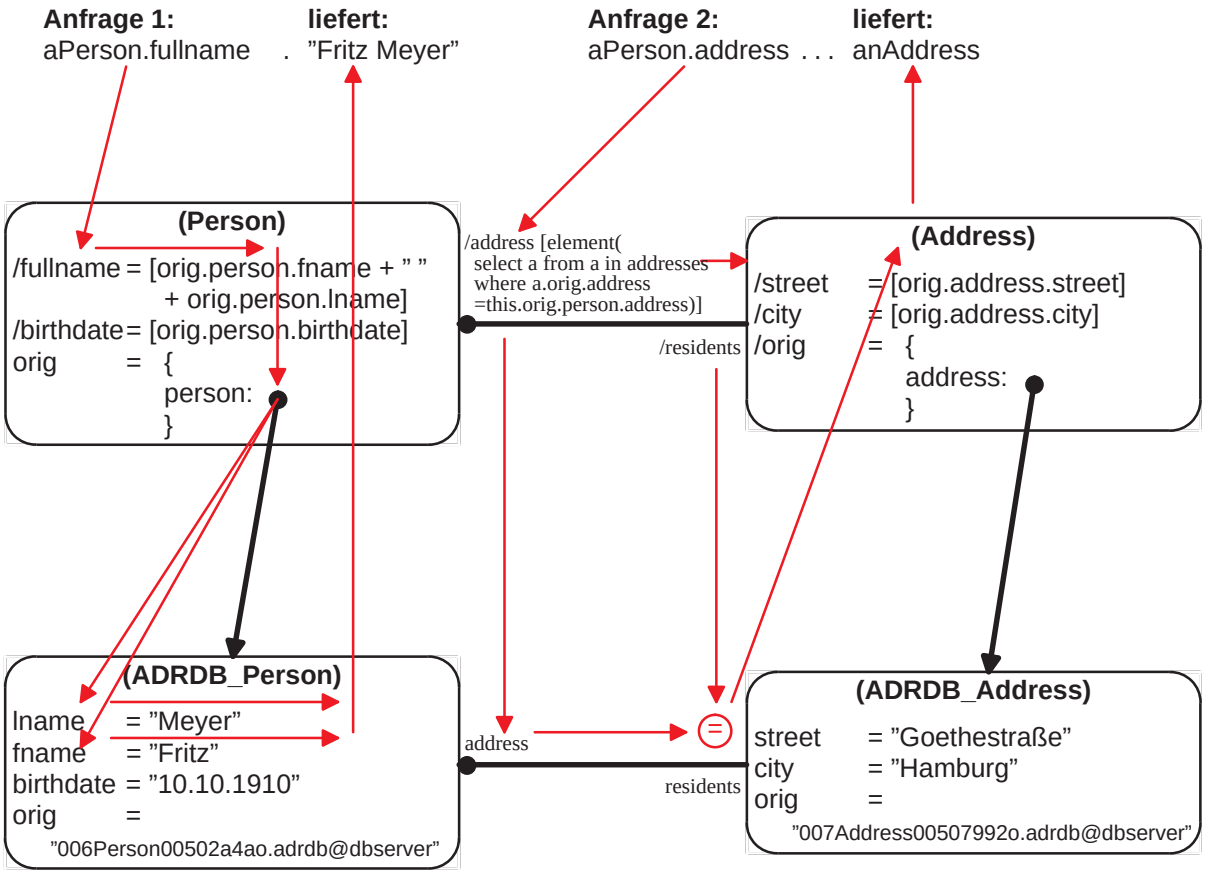


Bild 7: Beziehungen zwischen Konvertierungs- und Importklassen

3.4.2.2 Eingebettete Spezifikation von Importklassen

Importklassen unterscheiden sich sowohl in ihrer Struktur als auch in ihrer Herleitung von den Konvertierungsklassen. Da externe Objektreferenzen nicht direkt unterstützt werden, wird der Rückverweis durch eine Zeichenkette ausgedrückt, die die globale OID des externen Objekts enthält. Diese GOID wird zum Zeitpunkt der Objekterzeugung auch verwendet, um die spezifischste Klasse für dieses Objekt innerhalb der Vererbungshierarchie zu bestimmen (siehe Kapitel 4.3). Weiterhin werden alle Datenfelder im Importobjekt zwischengespeichert, um die Netzwerkzugriffe zu minimieren. Durch die reine 1-zu-1-Abbildung kann die Spezifikationssyntax wesentlich vereinfacht werden. Das folgende Beispiel zeigt die allgemeine Syntax.

Beispiel 3.3:

Spezifikation einer Importklasse in ODL*; die Erweiterungen sind unterstrichen:

```
interface mytype : supertype_list
    // type properties
    ( extent extentname imports extType from extDB::extExtent [[o] where condition];
      keys property_list ) : virtual
    {
    // instance properties
    [ virtual ] attribute     aType          attname;
    ...
    [ virtual ] relationship OtherType    relname inverse OtherType::invrel;
    ...
    // instance methods
    long        compute_value ( in long increment ) raises ( ExOverflow );
    ...
    };
```

■

Die angegebene Syntax unterscheidet sich in folgenden Punkten von der Spezifikation der Konvertierungsklassen:

- Andere Wurzelklasse

  Alle Importklassen müssen von der gemeinsamen Wurzelklasse ImportClass abgeleitet werden. Für die weitere Vererbungshierarchie gilt die Konsistenzbedingung von Seite 47.

- Keine Angabe von orig

  Die Referenz auf das externe Basisobjekt wird als GOID-Zeichenkette im orig-Attribut abgelegt. Da dies für alle Importklassen gilt, ist orig bereits in der Wurzelklasse ImportClass definiert und muß nicht mehr angegeben werden.

- Spezielle Extentklausel

  Um den Zugriff auf eine externe Datenbank ausdrücken zu können, wird für die Definition des Extents eine spezielle Syntax verwendet. Der Ausdruck spezifiziert die externe Datenbank extDB, die externe Klasse extType und den zugehörigen Extent extExtent. Die Importklasse wird instanziiert, indem auf den externen Extent zugegriffen und für jedes darin enthaltene Objekt eine Sichtinstanz mit der entsprechenden GOID als orig-Attribut erzeugt wird. Durch ein optionales Selektionsprädikat kann die Menge der importierten Objekte eingeschränkt werden; eine gegebenenfalls notwendige Laufvariable kann vor **where** angegeben werden. Im Gegensatz zu Konvertierungsklassen definiert jede Importklasse ihren Extent mit Hilfe der Basisklasse, auch wenn sie Subklasse einer anderen Importklasse ist. Der externe Klassenname wird in diesem Zusammenhang verwendet, um für eine gegebene GOID die korrekte Importklasse in der Hierarchie zu finden (siehe Kapitel H.6.6). Darüber hinaus wird er zur Anfrageoptimierung benötigt (siehe [FSF95]).

- Keine Datenfeldklauseln

  Da alle Datenfelder 1-zu-1-Kopien der Originalfelder sind, entfallen die Herleitungsklauseln. Namen und Typen der Datenfelder müssen mit den Originalfeldern übereinstimmen. Originalfelder, deren Namen nicht in der Spezifikation erscheinen, werden nicht importiert.

- Keine transienten Datenfelder

  Außer orig ist jedes Datenfeld automatisch virtuell und muß in der externen Originalklasse vorhanden sein. Weitere Datenfelder können nicht hinzugefügt werden.

Beispiel 3.4:

Die folgende Spezifikation definiert eine Importklasse:

```
interface ADRDB_Person : ImportClass
    (   extent adrdb_persons imports Person from ADRDB::persons;
        key (lname,fname) ) : virtual
    {
    attribute     string          lname;
    attribute     string          fname;
    attribute     string          birthdate;
    relationship  ADRDB_Address   address     inverse ADRDB_Address::residents;
    };
```

Die Importklasse ADRDB_Person importiert die externe Klasse Person aus der Datenbank ADRDB. Das Präfix wird verwendet, um später eine integrierte Klasse namens Person erzeugen zu können. Da kein Selektionsprädikat angegeben ist, werden alle externen Objekte importiert; die Attribute lname, fname und birthdate sowie die Relationship address werden übernommen. Zielklasse für die Relationship ist ADRDB_Address, die die Adreßklasse aus ADRDB importiert. ■

3.4.3 Separate Spezifikation

Der obige Ansatz kann nur verwendet werden, wenn der verwendete ODL-Parser an die neue Syntax angepaßt werden kann. Wenn jedoch ein vorhandener Parser benutzt werden muß, der neben der Codegenerierung noch datenbankspezifische Operationen ausführt, so muß man dem Parser eine ODMG-konforme Schemaspezifikation liefern und die zusätzlichen Sichtinformationen nachträglich hinzufügen.

Eine solche Situation hatten wir in der ersten Phase des IRO–DB-Projekts. Auf der interoperablen Seite sollte Ontos als Arbeitsdatenbank für die Sichtobjekte und ihre Spezifikationen verwendet werden. Die verwendete ODMG-Erweiterung von Ontos ([GOP96]) erzeugte ausgehend von der ODL-Spezifikation Ontos-spezifischen Code und trug die Schemainformation direkt in die Datenbank ein. Da dieser Parser nicht erweitert werden konnte, mußten alle Definitionen in Standard-ODL erfolgen und das Sichtverhalten in einem zusätzlichen Schritt hinzugefügt werden.

Bei diesem Ansatz erfolgt also eine Zweiteilung der Spezifikation. Eine ODL-Datei enthält die ODMG-konforme, syntaktische Schemaspezifikation. Diese besteht aus einer transienten Klassendefinition mit allen Attributen, Relationships und Methoden. In einer separaten Mapping-Datei wird dann für jede Klasse die Herleitung des Extents und der Datenfelder definiert. Für jede Klasse gibt es hierfür einen Datenblock, der mit **mapping** eingeleitet wird. Für Konvertierungsklassen wird die Extentklausel nach **def_ext** angegeben, und die Attribut- und Relationshipklauseln nach **def_att** bzw. **def_rel**. Für Importklassen muß nur der externe Extent und gegebenenfalls das Selektionsprädikat angegeben werden.

Beispiel 3.5:

Formale Darstellung einer Mapping-Klausel für Konvertierungs- und Importklassen:

```
mapping convClass {
    def_ext   query;
    def_att   attname  as query;
    ...
    def_rel   relname  as query;
    ...
    };

mapping impClass
    imports extType from extDB::extExtent [[o] where condition];
```

■

Das nachfolgende Beispiel gibt die separate Spezifikation für die beiden oben bereits benutzten Klassen an.

Beispiel 3.6:

ODL-konforme Typspezifikation in Schema.odl:

```
typedef struct { ADRDB_Person person; } Person_OrigType;

interface Person : DerivedClass
    (   extent persons
        key fullname ) : transient
    {
    attribute     Person_OrigType  orig;
    attribute     string           fullname;
    attribute     string           birthdate;
    relationship  Address          address     inverse Address::residents;
    };

interface ADRDB_Person : ImportClass
    (   extent adrdb_persons
        key (lname,fname) ) : transient
    {
    attribute     string           lname;
    attribute     string           fname;
    attribute     string           birthdate;
    relationship  ADRDB_Address    address     inverse ADRDB_Address::residents;
    };
```

Separate Spezifikation der Herleitung in Schema.map:

```
mapping Person {
    def_ext                 select Person( orig: struct( person:p) )
                            from p in adrdb_persons;
    def_att   fullname      as this.orig.person.fname + " " + this.orig.person.lname;
    def_att   birthdate     as this.orig.person.birthdate;
    def_rel   address       as element(
                                select a
                                from a in addresses
                                where a.orig.address = this.orig.person.address);
    };

mapping ADRDB_Person
    imports Person from ADRDB::persons;
```

■

3.5 Deklarativer und prozeduraler Zugriff

Wie bereits in Abschnitt 3.4.1 beschrieben, muß man bei objektorientierten Sichten sowohl die deklarative als auch die prozedurale Sichtweise beachten. Die Sichtspezifikation basiert dabei auf dem deklarativen Ansatz: Alle Herleitungen werden als OQL-Anfragen formuliert, die vom Anfragesystem ausgewertet werden können. Hierdurch sind die Herleitungen einfach zu formulieren und können oftmals, wie im relationalen System, durch Anfrageexpansion ausgewertet werden. Hierbei werden in Benutzeranfragen Zugriffe auf Sichtelemente durch die entsprechenden definierenden Anfragen ersetzt. Die hierbei erzeugte erweiterte Anfrage ist zwar im allgemeinen ziemlich groß, kann aber meist in einem Optimierungsschritt stark vereinfacht werden. Im günstigsten Fall kann die Auswertung vollständig auf den Originalobjekten basieren, so daß auf eine Instanziierung der Sichtobjekte verzichtet werden kann. Eine solche (mengenorientierte) Benutzeranfrage kann also sehr gut mit Hilfe der deklarativen Spezifikation ausgewertet werden. Weitere Details zur Anfrageexpansion und zur Optimierung, speziell in Bezug auf externe Datenbankzugriffe, sind in [FSF95] und [KaW97] zu finden.

Beispiel 3.7:

Gegeben sei die folgende Benutzeranfrage:

```
select struct ( name:p.fullname, birthdate:p.birthdate )
from p in persons;
```

Setzt man hier die Extentklausel und die Attributklauseln ein, erhält man die folgende erweiterte Anfrage:

```
select struct ( name:p.orig.person.fname + " " + p.orig.person.lname,
                birthdate:p.orig.person.birthdate )
from p in    select Person( orig: struct( person:_p ) )
             from _p in adrdb_persons;
```

Die Auswertung basiert vollständig auf den Attributwerten der Basisobjekte, die Person-Objekte selbst sind für die Auswertung nicht mehr notwendig und können aus der Anfrage entfernt werden. Dies vereinfacht die Anfrage zu:

```
select struct ( name:_p.fname + " " + _p.lname, birthdate:_p.birthdate )
from _p in adrdb_persons;
```

■

Es gibt jedoch Situationen, in denen eine solche Anfrageexpansion nicht möglich ist, und zwar immer dann, wenn aus bestimmten Gründen ein direkter Zugriff auf instanziierte Sichtobjekte notwendig ist (siehe [BuF98], [BuF99]):

- Anfragen mit objektwertigen Ergebnissen

  Wenn eine Benutzeranfrage nicht nur Attributwerte, sondern die Sichtobjekte selbst als Ergebnis zurückliefern soll, müssen die Objekte instanziiert werden, um sie in die Ergebnismenge einfügen zu können.

- Methodenaufrufe auf Sichtobjekten

  In ODMG–93 sind Methoden immer innerhalb der Sprachbindung definiert. Es gibt also keinen generischen Prozessor, der Methoden direkt ausführen kann, sondern es muß immer eine konkrete Methodenimplementierung in der Sprachbindung aufgerufen werden. Diese setzt für ihre Ausführung jedoch voraus, daß das entsprechende Sichtobjekt instanziiert ist. Wie oben bereits erläutert, wurde wegen der geringen Mächtigkeit von OQL auf eine deklarative, also expandierbare Methodenherleitung verzichtet.

- Updates auf Sichtobjekten

  OQL bietet selbst keine Sprachelemente zur Ausführung von Updates. Alle Updates müssen über Methoden realisiert werden, deren Aufrufe, wie gerade beschrieben, die Instanziierung des Sichtobjekts erfordern.

- Direktzugriffe in der Sprachbindung

  Wird im Rahmen der Sprachbindung durch Anwendungsprogramme auf die Sichten zugegriffen, so wird dies auf Grund der Einbettung der Datenbankobjekte in die Sprachumgebung im allgemeinen durch direkte Manipulation der Instanzen und nicht durch Auswertung deklarativer OQL-Anfragen geschehen. Diese Einbettung in die Programmierumgebung ist ein ausdrückliches Ziel von ODMG–93 (siehe S. 77).

- Komplexe Herleitungen

  Die vorgestellte Definition virtueller Attribute mit OQL ist nur für solche Herleitungen möglich, die sich als geschlossener OQL-Ausdruck beschreiben lassen. Werden komplexere Herleitungen benötigt, so können diese durch Paare von Lese- und Schreibmethoden realisiert werden, denen die volle Sprachmächtigkeit der verwendeten Programmiersprache zur Verfügung steht. (Dies ist leider schon für bedingte Zuweisungen notwendig; siehe die Anmerkungen zu einem erweiterten **or**-Operator in Abschnitt 3.7.2.) Diese Kombination deklarativer und prozeduraler Sichtspezifikation erlaubt eine einfache Spezifikation, wo immer es möglich ist, ohne jedoch komplexere Herleitungen auszuschließen.

Um diese Anforderungen zu erfüllen, muß neben dem deklarativen Zugriff auf die Sicht auch ein direkter, prozeduraler Zugriff auf die Sichtobjekte gemäß der Sprachbindung möglich sein. Dies umfaßt die Berechnung des Extents, die Instanziierung der Sichtobjekte und ein Zugriffsverfahren auf die Objekte, das äquivalent zur deklarativen Herleitung ist. Die in ODMG–93 definierte Sprachbindung und die notwendigen Anpassungen für das Sichtsystem sind im nachfolgenden Implementierungskapitel beschrieben.

3.6 Anwendung virtueller Klassen in den Sichtszenarien

Mit den vorgestellten virtuellen Klassen kann man auf externe Objekte zugreifen, Schnittstellen modifizieren, Daten umwandeln und Klassen miteinander integrieren. Die bei der Entwicklung zugrunde liegende Hauptanwendung war dabei die Integration verteilter Datenbanken. Die beiden Hauptphasen der Integration, Import und Konvertierung, sind bereits aus der vorgestellten Klassifikation der Sichtklassen ablesbar. Die Importphase kann dabei noch in externen Zugriff und Caching, die Konvertierungsphase in Homogenisierung und Vereinigung unterteilt werden (siehe Kapitel 2.3.2). In dem hier vorgestellten Ansatz übernimmt eine Importklasse sowohl den externen Zugriff als auch das Caching für eine externe Klasse; der Grad der verwendeten Zwischenspeicherung kann dabei in der Implementierung variiert werden (siehe Abschnitt 4.2.5.2). In Kapitel 6.7 wird eine Übertragung des Ansatzes auf CORBA vorgestellt, bei dem diese beiden Phasen explizit getrennt werden. Die Modellierung von Homogenisierung und Vereinigung liegt im Ermessen des Designers. Verwendet eine in der Sicht definierte integrierte Klasse direkt die Importklassen als Basisklassen, erfolgen Homogenisierung und Vereinigung in einem Schritt. Die Spezifikation einer solchen Klasse kann hierdurch sehr komplex werden. In diesem Falle kann man den Entwurf vereinfachen, indem man zusätzliche Sichtklassen für die Homogenisierung einfügt, und dann die so erzeugten Sichtklassen integriert. Hierdurch werden die Phasen getrennt und ihre Komplexität verringert sich.

Mit Hilfe der virtuellen Klassen können, neben den in Kapitel 2.2 vorgestellten allgemeinen Einsatzgebieten, alle in Bild 6 gezeigten Szenarien realisiert werden. Wie Bild 8 zeigt, beschränken

sich die beiden Varianten des verteilten Objektzugriffs auf die Verwendung von Importklassen. Die Standardimplementierung von Importklassen verwendet dabei objektweises Caching, bei dem der vollständige Objektzustand beim ersten Zugriff auf ein Objekt zwischengespeichert wird (siehe Kapitel 4.2.5.2). Alternativ kann auch attributweises Caching verwendet werden, bei dem nur jeweils die verwendeten Attributwerte im Importobjekt gespeichert werden. Die Verwendung direkter Propagierung, bei der alle Objektzugriffe direkt an die externen Klassen weitergeleitet werden, ist bei der Standardimplementierung nicht vorgesehen, kann aber durch Anpassung der Importklassen leicht erreicht werden. In diesem Fall enthalten die angelegten Proxy-Objekte neben der OID keine weiteren Attributfelder. Die in Abschnitt 6.7 vorgestellte Übertragung auf CORBA trennt die Cache-Funktionalität vollständig vom externen Zugriff und stellt beides als getrennte Klassen dar.

Neben diesen integrationsorientierten Anwendungen ist das dargestellte Sichtsystem auch für andere Aufgaben einsetzbar, in denen Sichten zum Einsatz kommen, wie etwa Schema-Anpassungen an Benutzer oder die Modellierung von Zugriffssicherheit (siehe Kapitel 2.2). Da die Definition einer Sicht neue Klassen erzeugt und diese dem Laufzeitsystem hinzugefügt werden müssen (Compilieren und Binden), ist dieses Sichtsystem vorzugsweise für gepflegte Sichten geeignet, die von einem Systemadministrator erstellt und betreut werden. Für interaktive ad-hoc-Sichten durch Endbenutzer ist das Verfahren eher ungeeignet. Weiterhin erfordern datenbankin-

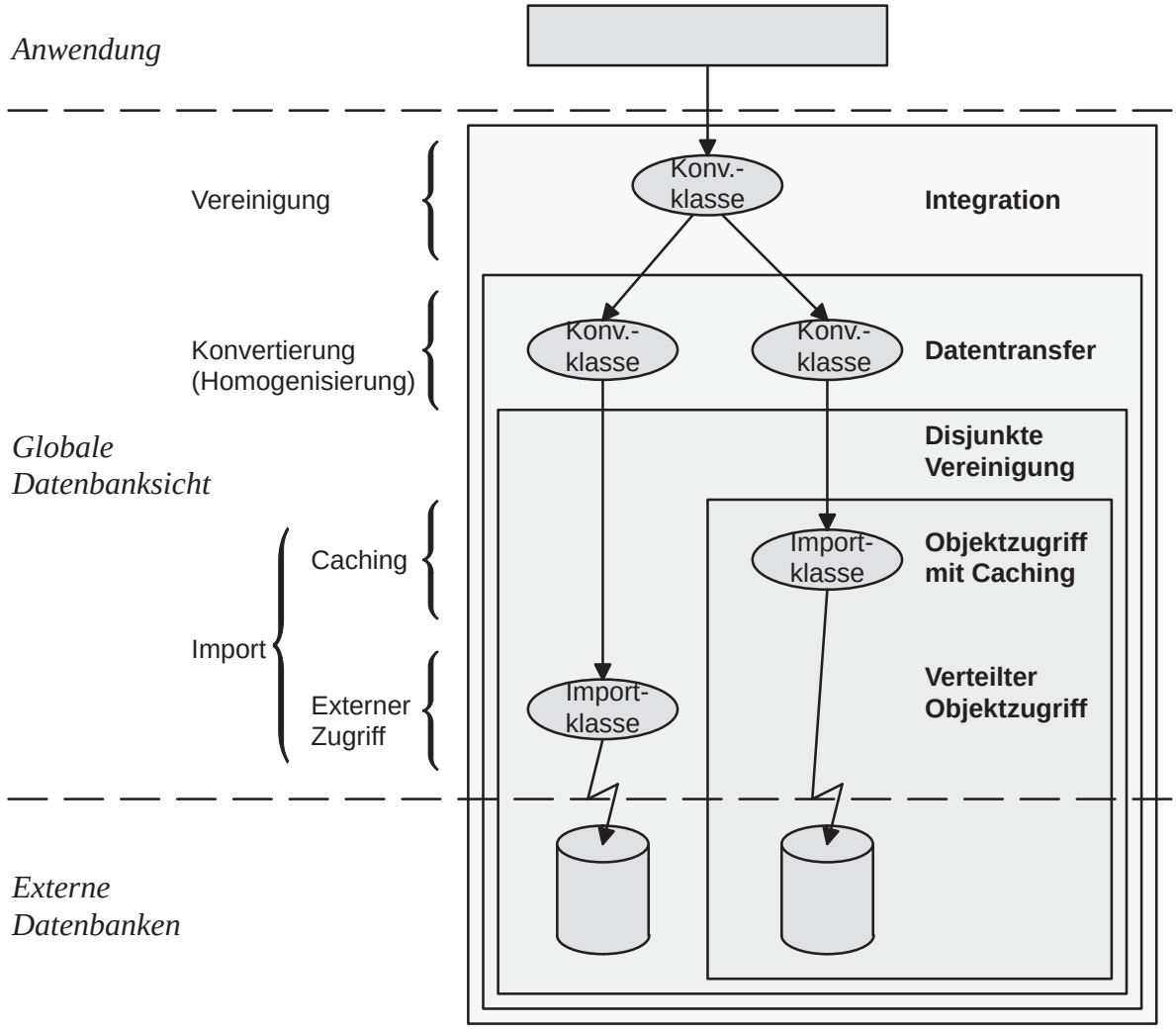


Bild 8: Variabler Integrationsgrad mit Import- und Konvertierungsklassen

terne Anwendungen, wie zum Beispiel Versionierung oder Schemaevolution, eine enge Einbindung des Sichtsystems in das verwendete Datenbanksystem, die möglicherweise von diesem als „Aufsatz“ konzipierten Sichtsystem nicht geleistet werden kann.

3.7 Anmerkungen zur Spezifikation

Im Rahmen des IRO–DB-Projekts sind verschiedene Varianten für Spezifikation, Auswertung und Konsistenzerhaltung dieser Sichten entwickelt worden. Einige interessante Zwischenergebnisse sind dabei erwähnenswert.

3.7.1 Ersatzkonstrukte für einen Outer Join

Der von Codd definierte *Outer Join* ([Cod79]) ist ein wichtiges Hilfsmittel bei der verlustfreien Integration relationaler Datenbanken und ist direkt auf objektorientierte Datenbanken übertragbar. Als Erweiterung des natürlichen Joins werden auch die Objekte in das Ergebnis übernommen, die kein korrespondierendes Objekt in der jeweils anderen Klasse haben; die dadurch fehlenden Attribute werden mit Nullwerten belegt. Bei der Integration sind beide Varianten notwendig: der natürliche Join liefert die Schnittmenge zweier Originalklassen, der Outer Join die Vereinigungsmenge — jeweils mit Unifikation der gemeinsamen Objekte. Für den Outer Join ist in SQL das Schlüsselwort **outer** vorgesehen, das aber nicht von allen Datenbanken unterstützt wird. Da es auch in ODMG–93 nicht definiert ist, muß man ein Ersatz-Konstrukt verwenden. Weitere Varianten des *(Full) Outer Join* sind der *Left Outer Join* und *Right Outer Join*, bei dem nur die korrespondenzfreien Objekte des linken beziehungsweise rechten Arguments dem natürlichen Join hinzugefügt werden.

Beispiel 3.8:

Mit einem natürlichen Join wird die Vereinigungsklasse Person nur mit den Personen instanziiert, die sowohl in der Adreß- als auch in der Autorendatenbank abgelegt sind:

```
interface Person : DerivedClass ( extent persons is
        select Person ( orig: struct( person:p, author:a ) )
        from p in adrdb_persons, a in pubdb_authors
        where p.lname = a.lastname and p.fname = a.firstname;
        [...] };
```

Bei einem (Full) Outer Join kommen noch die Personen hinzu, die nur in jeweils einer Datenbank erscheinen. Das jeweils andere Feld im Konstruktor (person bzw. author) wird auf **nil** gesetzt:

```
interface Person : DerivedClass ( extent persons is
        select outer Person ( orig: struct( person:p, author:a ) )
        from p in adrdb_persons, a in pubdb_authors
        where p.lname = a.lastname and p.fname = a.firstname;
        [...] };
```

Da ODMG–93 keinen Outer Join anbietet, muß man das folgende Ersatzkonstrukt verwenden, das zu dem gleichen Ergebnis führt:

```
interface Person : DerivedClass ( extent persons is
        select Person ( orig: struct( person:p, author:a ) )
        from p in adrdb_persons, a in pubdb_authors
        where p.lname = a.lastname and p.fname = a.firstname
    union
        select Person( orig: struct( person:p, author:nil ) )
        from p in adrdb_persons
        where not exists a in pubdb_authors: (p.lname = a.lastname and p.fname = a.firstname);
    union
        select Person( orig: struct( person:nil, author:a ) )
        from a in pubdb_authors
        where not exists p in adrdb_persons: (p.lname = a.lastname and p.fname = a.firstname);
        [...] };
```

Im vollständigen Beispiel in Anhang C wird die Klasse über einen Left Outer Join definiert, der zwar alle Personen aus der Adreßdatenbank übernimmt, korrespondenzfreie Autoren aber ignoriert. ■

3.7.2 Umgang mit unvollständigen Daten

Eine analoge Situation ergibt sich beim Zugriff auf Attribute. Während man zum Beispiel eine Summen- oder Maximumbildung als geschlossenen OQL-Ausdruck darstellen kann (V_1+V_2, max(set(V_1,V_2))), sind bedingte Ausdrücke nicht darstellbar (if V_2>0 then V_1+V_2 else V_1).[17] Hierzu gehört auch der Fall, daß ein Wert nur dann aus Datenbank 2 genommen werden soll, wenn der zugehörige Wert aus Datenbank 1 ein Nullwert ist. Einen solchen Fall gab es im Anwendungsszenario von IRO–DB, bei dem die Daten aus der einen Datenbank aktueller, aber nicht vollständig waren, und daher gegebenenfalls aus der zweiten Datenbank geholt werden mußten. In [BFN94] habe ich zu dem Zweck den in OQL enthaltenen **or**-Operator verallgemeinert, indem seine *Shortcut*-Semantik auf beliebige Werte erweitert wird. Ein Ausdruck a **or** b **or** c wird von links nach rechts ausgewertet und liefert den ersten Wert, der nicht **nil** ist. In Pseudocode läßt sich das so ausdrücken:

```
if (a != nil) then return a
else if (b != nil) then return b
else return c;
```

Dies ermöglicht eine elegante Handhabung unvollständiger Information. Eine noch größere Flexibilität erhält man durch Einführung des C++-Konstrukts für bedingte Ausdrücke: cond ? trueval : falseval, bei dem abhängig vom Wert der Bedingung cond entweder der linke oder der rechte Term ausgewertet wird. Der erweiterte **or**-Operator läßt sich hiermit ebenfalls ausdrücken, ist aber schwerer zu lesen:

```
a or b or c   ==    (a != nil) ? a : ( (b != nil) ? b : c )
```

Zu beachten ist, daß ODMG–93 zwar Nullwerte im generischen Teil definiert, diese in der C++-Sprachbindung aber nur für Objekte und nicht für Werte zur Verfügung stellt. Braucht man auch Nullwertfunktionalität für Werte, muß man entweder mit Konventionen arbeiten (zum Beispiel negatives Alter = unbekannt), oder die Gültigkeitsinformation explizit in Flags modellieren.

---

17. In [D4–2/1] habe ich eine Möglichkeit angegeben, um bedingte Ausdrücke mit Hilfe einer komplexen Mengenoperation mit geschachtelten Anfragen zu formulieren. Diese Konstruktion ist aber sehr künstlich und nicht handhabbar.

3.7.3 Sichere Dereferenzierung

Eine weitere sinnvolle Erweiterung in Bezug auf den Umgang mit Nullwerten ist eine sichere Dereferenzierung. Laut Standard soll beim Versuch, **nil** zu dereferenzieren, eine Ausnahme erzeugt werden. Diese kann in einer OQL-Anfrage aber nicht abgefangen und ausgewertet werden. Statt dessen bietet sich analog zum **or**-Operator eine Auswertungssemantik für die Zugriffsoperatoren . und → an, die **nil** zurückliefert, wenn das zu dereferenzierende Objekt **nil** ist. Der Ausdruck

```
aPerson.orig.person.lname or aPerson.orig.author.lastname
```

liefert also immer dann den Nachnamen aus dem Autorenpfad, wenn entweder im Personenpfad kein Nachname gespeichert ist, oder es überhaupt keine Person als Basisobjekt gibt. In reinem OQL läßt sich diese Semantik nicht ausdrücken, mit bedingten Ausdrücken wird er unleserlich:

```
(aPerson != nil and aPerson.orig.person != nil and aPerson.orig.person.lname != nil)
?   aPerson.orig.person.lname
:   ((aPerson != nil and aPerson.orig.author != nil and aPerson.orig.author.lastname != nil)
    ? aPerson.orig.author.lastname
    : nil )
```

3.7.4 Alternative Relationship-Spezifikation

Bei der Definition abgeleiteter Relationships in einer Sicht muß man auf eine Reihe von Konsistenzbedingungen achten.

a) Kardinalität:
   Bei 1-zu-1- und 1-zu-n-Relationships muß zugesichert werden, daß die einwertige Seite auch wirklich nur jeweils ein Element enthält, und nicht durch die Integration inkonsistenter Information unterschiedliche Objekte in derselben Relationship auftauchen.

b) Abgeschlossenheit:
   Die Zielobjekte der Relationship müssen in der als Ziel angegebenen Sichtklasse liegen. Ziele aus der Original-Relationship können eventuell nicht in die Sicht übernommen worden sein.

c) Konsistenz der Inversen:
   Beide Einzelpfade der Relationship müssen miteinander konsistent sein.

In der hier vorliegenden Arbeit werden Relationships durch einfache Selektionsanfragen auf der Zielklasse definiert. Die Abgeschlossenheit ist also in jedem Falle gesichert. Die Korrektheit der anderen beiden Kriterien muß durch korrekte und konsistente Definition der Anfragen zugesichert werden. Da dies aus Komplexitätsgründen nicht immer möglich ist, werden in meinem Sichtsystem nur bestimmte Relationships zugelassen:

Definition 3.11: Konsistenzbedingung für abgeleitete Relationships
   Eine Relationship zwischen zwei Sichtklassen, die nur von *einer* Basis-Relationship abgeleitet ist, behält die Kardinalität der Basis-Relationship bei. Eine Relationship zwischen zwei Sichtklassen, die von *mehr* als einer Basis-Relationship abgeleitet ist, erhält grundsätzlich die allgemeinste Kardinalität n-zu-m.

Im ersten Fall wird die Korrektheit der abgeleiteten Relationship direkt von der Basis-Relationship zugesichert, im zweiten Fall kann die Konsistenz leicht durch Vereinigung oder Durch-

schnitt der Basis-Relationships erreicht werden. Für eine konsistente Definition der beiden Zugriffspfade einer Relationship bietet sich eine Unterstützung durch das Entwurfswerkzeug an, indem man nur eine Herleitung für die gesamte Relationship angibt, aus der dann die beiden Herleitungsklauseln für die Pfade automatisch generiert werden.

In [D4–2/1] habe ich einen allgemeineren Ansatz verfolgt, der mehr Möglichkeiten bei der Integration von Relationships zuläßt. Sowohl die Datenmischung als auch die Konsistenzüberprüfung werden in den zugehörigen Anfragen modelliert. Da sich hierbei recht komplexe Anfragen ergeben, die redundant bei beiden Pfadausdrücken auftauchen, wird die Spezifikation in mehrere Teilanfragen aufgeteilt. Jede Relationship wird durch eine eigene benannte Anfrage definiert, die alle Objektpaare dieser Relationship aufsammelt. Die einzelnen Pfade dieser Relationship werden dann durch Selektion auf dieser Anfrage definiert. Die Anfrage selbst enthält zwei Phasen: das Mischen der Basis-Relationships und das Zusichern der Konsistenz. Um dabei die verschiedenen Kombinationsmöglichkeiten ausdrücken zu können, werden die Beiträge der einzelnen Basis-Relationships ebenfalls in eigenen Anfragen spezifiziert. Diese Einzelanfragen können dann vereinigt, geschnitten oder beliebig anders kombiniert werden, um danach all die Einträge wieder zu entfernen, die die Konsistenz des Ergebnisses beeinträchtigen.

Der Beitrag einer einzelnen Datenbank DB zu einer integrierten Relationship REL kann durch folgende Formel ausgedrückt werden:

```
define REL_from_DB as
    select distinct struct ( left:l, right:r )
    from l in leftClass, r in rightClass
    where (l.orig.DB != nil) and (r.orig.DB != nil) and
          ( (r.orig.DB Θ1 l.orig.DB.REL) or (l.orig.DB Θ2 r.orig.DB.INVREL) );
```

Hierbei sind die generischen Operatoren Θ_1 und Θ_2 folgendermaßen definiert:

Θ_1 = „=“, Θ_2 = „=“ für 1-zu-1-Relationships
Θ_1 = „**in**“, Θ_2 = „=“ für 1-zu-n-Relationships
Θ_1 = „**in**“, Θ_2 = „**in**“ für n-zu-m-Relationships

Auf Grund der Symmetrie von Relationship und inverser Relationship muß nur ein Term der letzten Klammer angegeben werden, also entweder Θ_1 oder Θ_2. Das Ergebnis dieser Anfrage ist die Menge aller integrierten Objektpaare, deren Basisobjekte in Datenbank DB durch die Relationship REL verbunden sind. Durch die Vereinigung aller so definierten Teilanfragen erhält man die Menge aller potentiellen Verbindungen. Falls die Datenbanken jedoch widersprüchliche Informationen enthalten, so kann sich bei der Vereinigung die falsche Kardinalität ergeben. Um dies zu vermeiden, können durch die folgenden drei Filterregeln alle widersprüchlichen Einträge gelöscht werden. Durch weitere Filter können auch andere Lösungsstrategien implementiert werden; bei einer Schnittbildung sind solche Filter nicht notwendig.

```
define one_to_one_template as
    select distinct rel
    from rel in ( source1 union source2 union ... )
    where not exists other in ( source1 union source2 union ... ) :
        (rel != other) and ( (rel.left = other.left) or (rel.right = other.right) );

define one_to_many_template as
    select distinct rel
    from rel in ( source1 union source2 union ... )
    where not exists other in ( source1 union source2 union ... ) :
        (rel != other) and (rel.left = other.left);

define many_to_many_template as
    select distinct rel
    from rel in ( source1 union source2 union ... );
```

Die Relationship-Pfade in den beiden beteiligten Klassen werden dann durch Selektion auf der so erzeugten Relationship-Anfrage definiert. Beim Zugriff auf einen solchen Pfad werden die geschachtelten Anfragen expandiert und soweit wie möglich vereinfacht.

Beispiel 3.9:

Gegeben seien zwei Adreßdatenbanken ADRDB1 und ADRDB2, die beide das in Anhang C definierte Schema mit den Klassen Person und Address zur Verfügung stellen. Die beiden Klassen von ADRDB1 seien als Person1 und Address1, die Klassen von ADRDB2 als Person2 und Address2 importiert worden. Zwei neue Klassen, Person und Address sollen beide Datenbanken, inklusive der Relationship integrieren. Die beiden orig-Attribute haben die folgende Struktur:

```
typedef struct { Person1 person1; Person2 person2; } Person_OrigType;
typedef struct { Address1 address1; Address2 address2; } Address_OrigType;
```

Auf eine Angabe der vollständigen Klassendefinition sei hier verzichtet. Als erster Schritt erfolgt die Definition von Teilanfragen zur Bestimmung der Beiträge der einzelnen Datenbanken. Nach dem obigen Schema ergibt dies:

```
define RelPersonAddress_from_ADRDB1 as
    select distinct struct ( person:p, address:a )
    from p in persons, a in addresses
    where (p.orig.person1 != nil) and (a.orig.address1 != nil) and
          (a.orig.address1 = p.orig.person1.address);

define RelPersonAddress_from_ADRDB2 as
    select distinct struct ( person:p, address:a )
    from p in persons, a in addresses
    where (p.orig.person2 != nil) and (a.orig.address2 != nil) and
          (a.orig.address2 = p.orig.person2.address);
```

Bei der Vereinigung dieser Teilanfragen kann es zu einer falschen Kardinalität kommen (n-zu-m statt 1-zu-n), wenn nämlich für eine Person unterschiedliche Adressen gespeichert sind. Mit dem entsprechenden Filterausdruck ergibt sich für die Relationship RelPersonAddress die folgende Definition:

```
define RelPersonAddress as
    select distinct rel
    from rel in ( RelPersonAddress_from_ADRDB1 union RelPersonAddress_from_ADRDB2 )
    where not exists
        other in ( RelPersonAddress_from_ADRDB1 union RelPersonAddress_from_ADRDB2 ):
        (rel != other) and (rel.person = other.person);
```

Die beiden Pfade in den Klassen Person und Address werden folgendermaßen durch Selektion auf dieser Relationship-Anfrage definiert.

```
interface Person : DerivedClass (...) { ...
    relationship  Address          address inverse Address::residents is
                                       element (select rel.address
                                                from rel in RelPersonAddress
                                                where rel.person = this);
    ...};

interface Address : DerivedClass (...) { ...
    relationship  set<Person>    residents inverse Person::address is
                                       select rel.person
                                       from rel in RelPersonAddress
                                       where rel.address = this);
    ...};
```

■

Diese Art der Spezifikation führt zu sehr komplexen Anfragen, die kaum effizient ausgeführt werden können. Häufig ist eine vollständige Instanziierung beider beteiligten Klassen notwendig, um die Konsistenz der Relationship testen zu können. Aus diesem Grunde ist der Ansatz zu Gunsten der oben angegebenen eingeschränkten Version fallen gelassen worden.

4 Implementierung

Die im vorhergehenden Kapitel beschriebenen Sichten definieren nur das Modell für ein Datenbanksystem bzw. eine Sichtimplementierung. Für eine Implementierung muß dieses Modell auf eine konkrete Repräsentation in der zu verwendenden Programmiersprache abgebildet werden. Die Basisdatentypen und Konstruktoren des Modells werden etwa auf die vordefinierten Datentypen und Konstruktoren der Programmiersprache abgebildet. Höhere Konstrukte, wie Klassen oder Tabellen, werden häufig nicht direkt von der Sprache unterstützt und müssen daher geeignet nachmodelliert werden. Hierzu gehört auch die Definition der Zugriffsfunktionen in der Sprache. Sowohl die Leistungsfähigkeit der verwendeten Programmiersprache als auch das angewandte Modellierungsverfahren bestimmen die Fähigkeiten der Datenbankimplementierung. Soll ein Datenbanksystem um Sichtfunktionalität erweitert werden, so müssen die Einschränkungen, die sich aus der gewählten Implementierung ergeben, berücksichtigt werden.

Dieses Kapitel beschreibt die konkrete Umsetzung des vorgestellten Ansatzes für eine C++-Datenbank. Nach einem allgemeinen Überblick über die Laufzeitumgebung in Abschnitt 4.1 werden einige Kernkomponenten im Detail vorgestellt: Abschnitt 4.2 beschreibt die sogenannte Sprachbindung, also die Darstellung der einzelnen Datenbankklassen in Form von C++-Objekten. In den Abschnitten 4.3 und 4.4 werden die Objektverwaltung mit ihrem Referenzmechanismus und das Verfahren zur Zusicherung referenzieller Integrität bei Relationships vorgestellt. Abschnitt 4.5 erläutert die Probleme, die sich aus der von ODMG geforderten direkten Einbettung in C++ im Zusammenhang mit der Anfrageauswertung ergeben und zeigt Lösungen für den generischen Zugriff auf kompilierte Strukturen. Im letzten Abschnitt wird schließlich das Zusammenspiel der vorgestellten Sichtkomponenten mit dem vollständigen IRO–DB-System beschrieben.

Das gesamte Sichtsystem ist im Rahmen des EU-Projekts IRO–DB implementiert und verifiziert worden. Die Implementierung erfolgte auf Sun Workstations unter SunOS 4.1.3. Verwendet wurde der C++ 3.0.1 Sun SPARCompiler (SC2.0.1) mit „Jumbo-Patch“ 100967-08.

Das Implementierungskapitel ist mit Absicht umfangreich gehalten, da es viele Detailprobleme vorstellt und löst, die bei der Umsetzung eines Sichtsystems auf statisch getypten, kompilierenden Objektbanken zwangsläufig auftreten. Der Nachweis der Implementierbarkeit des vorgestellten Sichtsystems stellt einen Schwerpunkt der Arbeit dar, da die meisten existierenden Ansätze die Einschränkungen bei ihrer Umsetzung völlig außer acht lassen (siehe Motivation und Verwandte Arbeiten). Zusätzliche Details, die den Umfang dieses Kapitels gesprengt hätten, sind in Anhang H zu finden.

4.1 Übersicht: Systemarchitektur

Das hier vorgestellte System ist in der Programmiersprache C++ implementiert worden. Es basiert auf der offiziellen ODMG-Sprachbindung für C++, die konkretisiert und um sichtspezifische Besonderheiten ergänzt wurde. Bild 9 zeigt den Kern der Laufzeitumgebung. Unter dem Anwendungsprogramm findet man vier Module: Die Anfragebearbeitung mit dem Data Dictionary, die Objektverwaltung, die Wrapperfunktionen und die Sprachbindung; darunter liegen die Netzwerkkomponente und die eigentlichen Datenbanken. Diese Module haben jeweils die folgende Funktionalität:

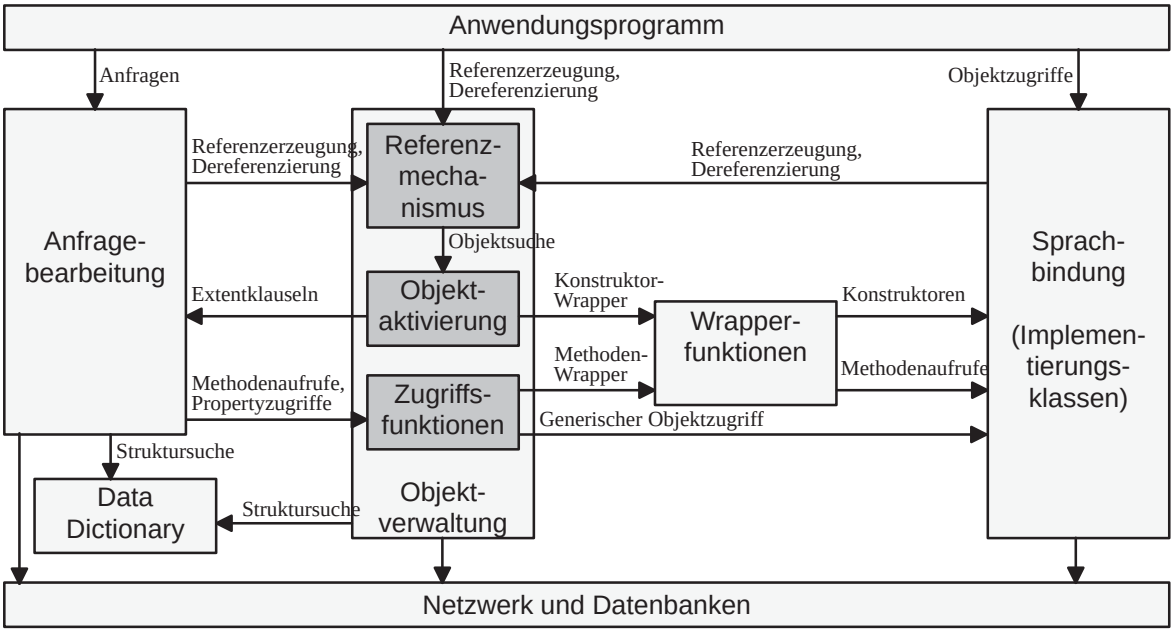


Bild 9: Architektur der Laufzeitumgebung

Sprachbindung (siehe Kapitel 4.2 und Kapitel 4.4)

Die Sprachbindung stellt die eigentliche Schnittstelle vom Anwendungsprogramm zur Datenbank zur Verfügung. Sie bildet die konzeptuellen Datenbankklassen auf C++-Klassen ab, auf die wie auf normale Klassen zugegriffen werden kann, die ihre Daten aber aus der Datenbank holen und dort wieder persistent ablegen. Die Sprachbindung besteht aus einem schemaspezifischen Teil, der die konkreten Klassen für das aktuelle Schema implementiert, und einem generischen Teil, der all die Klassen umfaßt, die von allen Schemata gemeinsam genutzt werden.

Anfragebearbeitung (siehe Kapitel 4.5)

Die Anfragebearbeitung stellt die zweite Schnittstelle zum System dar. Während die Sprachbindung programmatischen Zugriff auf Basis von kompiliertem C++ bietet, kann man mit der Anfragebearbeitung deklarative OQL-Anfragen auswerten. Das *Data Dictionary* stellt hierfür die notwendigen Schemainformationen zur Verfügung.

Wrapperfunktionen (siehe Kapitel 4.5)

Im Gegensatz zu relationalen Systemen, bei denen die gesamte Anfragebearbeitung auf den generischen Strukturen der Datenbank arbeiten kann, muß man bei einer objektorientierten Datenbank auch auf die schemaspezifischen Objekte der Sprachbindung zugreifen, um zum Beispiel Methoden ausführen zu können. Das Hauptproblem, das sich hierbei ergibt, ist die mangelnde Strukturinformation von kompiliertem C++-Code. Um zur Laufzeit generisch auf Komponenten der Sprachbindung zugreifen zu können, sind eine Reihe von sogenannten *Wrapperfunktionen* notwendig, die sowohl vom Anfragemodul als auch von der Objektverwaltung verwendet werden. Das Anfragemodul greift dabei nicht direkt auf die Wrapper zu, sondern über eine Reihe vordefinierter *Zugriffsfunktionen* der Objektverwaltung.

Objektverwaltung (siehe Kapitel 4.3)

Die Objektverwaltung bildet den eigentlichen Kern des Sichtsystems. Sie interagiert mit der Anfragebearbeitung, den Wrapperfunktionen, der Sprachbindung und der Datenbank. Sie ist dafür zuständig, angeforderte Objekte aus der Datenbank in die Sprachbindung zu holen und am Ende wieder zu deaktivieren (*Objektaktivierung*). Die besondere Herausfor-

derung für eine Objektverwaltung in einem identitätserhaltenden System ist dabei die Zusicherung einer konsistenten Objektzuordnung: Sichtobjekte dürfen nicht mehrfach erzeugt werden, sondern müssen nach ihrer erstmaligen Erzeugung in allen weiteren Anfragen wiederverwendet werden, um Objektidentität zwischen den Anfragen und der Sprachbindung zu gewährleisten. Ein wichtiger Aspekt ist hierbei ein guter *Referenzmechanismus*, der eine möglichst späte Objektaktivierung erlaubt.

Die generischen Teile, wie Anfragebearbeitung, Objektverwaltung und die Basisklassen der Sprachbindung werden als feste Funktionsbibliothek zur Verfügung gestellt. Die schemaspezifischen Teile der Sprachbindung und die Wrapperfunktionen werden mit Hilfe eines Präprozessors automatisch aus der Sichtspezifikation generiert (siehe Kapitel 4.6.1, Designumgebung). Das nachfolgende Beispiel zeigt ein typisches Szenario für die Zusammenarbeit der Module.

Beispiel 4.1:

Das Anwendungsprogramm stellt eine Anfrage an die Anfragebearbeitung. Diese holt sich alle notwendigen Schemainformationen aus dem Data Dictionary, um die Anfrage zu überprüfen, zu modifizieren und zu optimieren. Handelt es sich um eine reine Wertanfrage, so wird sie direkt an die Datenbank beziehungsweise an die Netzkomponente zur Auswertung weitergereicht. Soll die Anfrage Objekte zurückliefern, so werden die notwendigen Parameter, wie Klassenname, Oid oder Attributwerte, an die Objektverwaltung gegeben, die hieraus Objektreferenzen erstellt. Die Objekte selbst müssen zu diesem Zeitpunkt noch nicht aktiviert werden. Diese Referenzen werden dann als Anfrageergebnis an das Anwendungsprogramm zurückgegeben. Das Programm iteriert nun über die Antwortmenge und dereferenziert eine Referenz. Die Objektverwaltung versucht nun, das zugehörige Objekt der Sprachbindung zurückzuliefern. Falls dieses Objekt bereits aktiviert ist, muß es gefunden und direkt zurückgegeben werden. Ist es noch nicht aktiviert, werden die in der Referenz gespeicherten Daten (Klassenname, Oid, etc.) verwendet, um das zugehörige Objekt anzulegen, indem über eine Wrapperfunktion der zugehörige C++-Konstruktor aufgerufen wird. Sind für die Aktivierung weitere Parameter notwendig, so werden die notwendigen Klauseln aus dem Data Dictionary bestimmt und mit Hilfe der Datenbank und gegebenenfalls der Anfrageauswertung berechnet. Nach der Dereferenzierung kann das Anwendungsprogramm direkt auf dem Objekt der Sprachbindung arbeiten und zum Beispiel auf Attributwerte zugreifen oder Methoden aufrufen. Wird auf Relationships zugegriffen, welche ebenfalls mit Hilfe von Referenzen implementiert sind, so wird aus der Sprachbindung heraus wiederum die Objektverwaltung angesprochen, um das Zielobjekt zu erhalten. ■

4.2 Die Sprachbindung

In diesem Kapitel wird die Sprachbindung erläutert, also die Darstellung der konzeptuellen Datenbankklassen in der verwendeten Programmiersprache. Zuerst wird die Wahl von C++ als Implementierungssprache begründet, dann folgen die allgemeine Sprachbindung von ODMG–93 und schließlich die konkreten Erweiterungen für das Sichtsystem.

4.2.1 Sprachauswahl

Da sich diese Arbeit nur mit objektorientierten Sichten beschäftigt, betrachten wir nur die Implementierung von objektorientierten Datenbanksystemen. Es ist zwar möglich, und teilweise auch notwendig, objektorientierte Datenbanken in prozeduralen und funktionalen Sprachen zu implementieren, allerdings ist es naheliegender, auch eine objektorientierte Programmiersprache zur

Implementierung zu verwenden. Die wichtigsten Vertreter objektorientierter Programmiersprachen sind Smalltalk, C++ und Java.

Smalltalk

Smalltalk ([GoR85]) gilt mit seinem soliden, ‚echt' objektorientierten Konzept als Wegbereiter der Objektorientierung, konnte sich aber nie auf dem Markt etablieren. Auf Grund der prozeduralen Tradition der Firmen hat das System auch heute, trotz zunehmender Leistungsfähigkeit, kaum eine Chance, außerhalb von Komplettlösungen und im Forschungsbereich als universelle Programmiersprache eingesetzt zu werden. Eine weitere Einschränkung ist die Auslegung als abgeschlossenes Einzelplatzsystem mit separaten Arbeitsumgebungen (Images). Obwohl die Konzepte zum Datenaustausch immer mehr erweitert werden, bleibt eine gemeinsame, konkurrierende Datenhaltung schwer in das System zu integrieren. Dennoch ist mit GemStone ([SDC91]) eine objektorientierte Datenbank auf Smalltalk implementiert worden.

C++

Die zweite bedeutende objektorientierte Programmiersprache ist C++. Sie ist durch Erweiterung der Programmiersprache C um objektorientierte Konzepte entstanden und hat die älteren objektorientierten Pascal-Varianten fast völlig verdrängt. Ein Nachteil der Verwandtschaft zu C ist die großteils beibehaltene Kompatibilität zum C-Programmtext, was die Möglichkeiten der Sprache einschränkt und zu einem oft unsauberen Design der Sprachelemente führt. Weiterhin ist die Sprache nicht wirklich objektorientiert im Sinne des Manifesto ([ABW*89]). Es wird zwar mit Klassen und Methoden eine Gruppierung und gewisse Kapselung erreicht, die aber auch die privaten Teile der Klassendefinition offenlegt und somit unvollständig ist. Weiterhin verhalten sich die Basisdatentypen anders als benutzerdefinierte Objekte, und Typen sind selbst nicht als eigene Objekte zur Laufzeit zugreifbar. Auf der anderen Seite ist aber die enge Verwandtschaft mit C ein großer Vorteil von C++. Durch die hohe Verbreitung prozeduraler Sprachen am Markt hat sich C++ als ‚natürliche Weiterentwicklung' leicht durchsetzen können und beherrscht den objektorientierten Markt. Dies erkennt man auch daran, daß für wesentliche Standards Sprachbindungen für C++ definiert werden. Sowohl ODMG–93 als auch die CORBA-Architektur der Object Management Group ([OMG91]) sind Beispiele hierfür.

Java

Die neueste objektorientierte Sprache ist Java. Sie baut auf der Syntax von C++ auf, verwendet jedoch ein einfacheres und schlüssigeres Datenmodell. Mehrfachvererbung und Templates entfallen ebenso wie die maschinennahen Manipulationsmöglichkeiten (Zeiger, Speicheranforderung, Dereferenzierung usw.). Hinzu kommen neben Interfaces und einem Modulkonzept eine funktional reichhaltigere Laufzeitumgebung mit Garbage Collection, Metainformationen und einem Sicherheitskonzept. Der Hauptunterschied ist jedoch, daß Java nicht in Maschinencode übersetzt, sondern in Form eines Bytecodes interpretiert wird, wodurch eine hohe Portabilität und dynamische Erweiterbarkeit erreicht wird. Inzwischen ist Java als Modesprache für Internetanwendungen weit verbreitet, und auch CORBA und ODMG definieren entsprechende Sprachbindungen.

Andere Sprachen

Andere objektorientierte Programmiersprachen, wie etwa Eifel oder verschiedene Objekt-Pascal-Varianten, haben nie die notwendige Verbreitung erreicht. LISP ist noch relativ nah an

objektorientierter Programmierung, wird aber hauptsächlich für Dokumentenverarbeitung (siehe Emacs oder Interleaf) oder für Anwendungen der Künstlichen Intelligenz verwendet. Das objektorientierte Datenbanksystem O_2 ist zwar in LISP implementiert, stellt aber eher einen Sonderfall dar. Die Sprache C ist hervorragend zur Systemprogrammierung geeignet, ist aber nicht objektorientiert; C++ als C-Erweiterung bietet hier wesentlich mehr Konzepte und eine bessere Typsicherheit.

Auf Grund der Bedeutung von C++ geht die vorliegende Arbeit speziell auf die Randbedingungen einer Sichtimplementierung in C++ ein. Java war zu Beginn der Arbeit noch in der Entwicklung und kaum verbreitet. Auf eine Anpassung der vorgestellten Implementierung an Java wird daher gesondert in Kapitel 6.1 eingegangen.

4.2.2 Möglichkeiten zur Objektmodellierung in C++

Die Sprachbindung definiert die Beziehung zwischen konzeptuellen Datenbankobjekten und instanziierten Implementierungsobjekten der Programmiersprache.

Definition 4.1: Datenbankobjekt
Ein *Datenbankobjekt* ist ein Objekt im Sinne von Definition 2.3, also ein konzeptuelles Objekt im ODMG-Datenmodell. Dies umfaßt sowohl persistente als auch virtuelle Objekte.

Definition 4.2: Implementierungsobjekt
Ein *Implementierungsobjekt* ist ein Objekt der Programmiersprache, also eine Instanz einer C++-Klasse. Ein Datenbankobjekt kann durch ein oder mehrere Implementierungsobjekte dargestellt werden.

Implementierungsobjekte ermöglichen den Anwendungsprogrammen den Zugriff auf die Datenbank; sie repräsentieren die Datenbankobjekte in der Programmiersprache und stellen die notwendigen Zugriffsoperationen zur Verfügung. Bei dieser Abbildung muß man zwischen zwei grundsätzlichen Verfahren unterscheiden. Bei der *direkten* Abbildung wird für jedes Datenbankobjekt ein äquivalentes C++-Objekt als Implementierungsobjekt erzeugt, bei der *indirekten* Abbildung wird ein Datenbankobjekt aus einer variablen Menge generischer C++-Objekte zusammengesetzt. Beide Verfahren unterscheiden sich hinsichtlich ihrer Effizienz und der verfügbaren Funktionalität und werden im folgenden detailliert beschrieben.

4.2.2.1 Direkte, einbettende Abbildung

Die erste Variante ist die direkte Einbettung des Datenbankschemas in das C++-Datenmodell. Die Grundidee ist, daß jede Datenbankklasse durch eine strukturell gleiche C++-Klasse repräsentiert wird. In dieser 1-zu-1-Abbildung werden die Attribute der Datenbank durch Datenfelder im C++-Objekt repräsentiert und Operationen durch virtuelle C++-Methoden. Die Vererbungshierarchie der Datenbankklassen wird dabei ebenfalls in die Vererbungshierarchie der C++-Klassen übernommen. Jede Instanz einer Datenbankklasse führt zu einem Objekt der zugehörigen C++-Klasse. Das C++-Objekt repräsentiert direkt das entsprechende Datenbankobjekt: Änderungen auf den Datenfeldern des Objekts sind Änderungen an den persistenten Attributen des Datenbankobjekts. Methodenaufrufe auf dem Objekt entsprechen den im Schema definierten Operationen.

Die Vorteile der direkten Abbildung liegen in der natürlichen Anwendung der Sprache. Der Anwender sieht keinen Unterschied zwischen persistenten und transienten Objekten und kann mit

Datenbankobjekten genau wie mit normalen Objekten der Sprache arbeiten. Attributzugriffe und Methodenaufrufe auf Datenbankinstanzen unterscheiden sich nicht von den gewöhnlichen Objektzugriffen unter C++. Außerdem arbeitet dieses Verfahren sehr effizient, da die Objekte nur wenig Speicher beanspruchen und der Compiler sich um schnelles Laufzeitverhalten kümmert. Der wesentliche Nachteil dieses Ansatzes ist die Beschränkung auf die Funktionalität des C++-Systems. Sowohl der Typumfang als auch die Laufzeitunterstützung sind durch den Sprachstandard und den Compiler beschränkt. Dies macht sich besonders bei der Anfragebearbeitung bemerkbar. Eine genaue Beschreibung der Probleme folgt weiter unten. In Beispiel 4.2 auf Seite 73 ist exemplarisch eine direkte Abbildung angegeben.

4.2.2.2 Indirekte, aufsetzende Abbildung

Bei der indirekten Abbildung werden die Datenbankklassen nicht direkt in das C++-Typsystem integriert sondern in einer generischen Metadarstellung nachmodelliert. Hierdurch kann man die Einschränkungen durch das C++-Typsystem umgehen und man hat die Möglichkeit, zusätzliche Funktionalität anzubieten. Eine Datenbankklasse wird hier nicht in eine gleichstrukturierte C++-Klasse übersetzt, sondern es wird eine feste Menge vordefinierter C++-Klassen verwendet, mit deren Hilfe die Datenbankklasse modelliert wird. Für die verschiedenen Elemente eines Schemas stehen dabei verschiedene C++-Klassen zur Verfügung. Ein atomares Attribut kann zum Beispiel durch eine Instanz einer C++-Klasse namens SimpleAttribute dargestellt werden. Die Instanz enthält dann sowohl den Namen des Attributs als auch den Attributwert und eine Beschreibung seines Typs. Zur Darstellung strukturierter Attribute kann man etwa eine Klasse namens StructuredAttribute verwenden, die intern eine verkettete Liste von Datenfeldobjekten verwaltet. Bei einer solchen Abbildung wird ein Datenbankobjekt nicht auf ein einzelnes Implementierungsobjekt abgebildet, sondern auf ein komplettes Netzwerk aus Implementierungsobjekten verschiedenen Typs. Die verwendeten Klassen sind vollständig vordefiniert und enthalten daher keinen schemaabhängigen Code. Folglich können auch die Methoden der Datenbankobjekte nicht als Member-Funktionen definiert werden, sondern müssen als freie C++-Funktionen implementiert werden, die als zusätzlichen Parameter entweder das Wurzelobjekt eines solchen Objektnetzwerks, eine OID oder eine Referenz übergeben bekommen. Ein Beispiel für eine indirekte Abbildung ist im Datenbanksystem VODAK zu finden ([Kla93]).

Der Vorteil dieses Verfahrens ist die Unabhängigkeit von den C++-Typrestriktionen. Außerdem kann man zur Laufzeit Datentypen verändern und neu definieren; man kann neue Schemata zum System hinzufügen, ohne neu kompilieren zu müssen; Objekte können ihre Klassenzugehörigkeit und Struktur verändern; das Methoden-Dispatching kann den Bedürfnissen der Datenbank angepaßt werden und so weiter. Der Nachteil bei diesem Verfahren ist, daß Datenbankobjekte anders als ‚normale' Objekte behandelt werden müssen. Man kann nicht mehr die in C++ übliche Punktnotation für den Attributzugriff verwenden, sondern muß hierfür spezielle Funktionen aufrufen. Das gleiche gilt für Methodenaufrufe. Eine Möglichkeit, dies zu umgehen, ist die Bereitstellung einer speziellen Datenmanipulationssprache, die die gewünschte Syntax wieder zur Verfügung stellt und durch einen Präprozessor oder Interpreter auf den entsprechenden C++-Code abgebildet wird (siehe etwa VODAKs VML). Hierbei erhält man aber wieder die von den relationalen Datenbanken bekannte Sprachtrennung zwischen Anwendungsprogramm und Datenbankzugriff und man verliert die Mächtigkeit der C++-Programmiersprache. In jedem Fall sind die Datenbankobjekte nicht mehr nahtlos in die Programmiersprache eingebettet und das System wird langsamer arbeiten, da bei Objektzugriffen durch das Objektnetzwerk navigiert werden muß. Dennoch ist der Gewinn in der Funktionalität so groß, daß man diese Nachteile durchaus in Kauf nehmen kann. Die indirekte Abbildung ist ebenfalls in Beispiel 4.2 anschaulich dargestellt.

4.2.2.3 Mischverfahren

Neben der ausschließlichen Verwendung eines der beiden Verfahren besteht auch die Möglichkeit, beide in einem Mischverfahren miteinander zu kombinieren. Hierbei kann man die Vorteile beider Systeme ausnutzen und eine Einbettung erreichen, die die großen Einschränkungen durch C++ vermeidet. Man stellt hierbei ein Klasseninterface zur Verfügung, das der direkten Abbildung entspricht. Für jedes Datenbankobjekt gibt es also wieder ein entsprechendes C++-Objekt, das das gleiche Interface anbietet. Für die interne Auswertung wird jedoch das Metaformat aus der indirekten Abbildung verwendet. Die Grenze zwischen Daten im direkten und indirekten Format kann dabei unterschiedlich gelegen sein. Bei einer stark indirekten Ausprägung sind die Interface-Objekte einfache Hüllen (*Wrapper*), die selbst keine Daten vorhalten sondern nur die Zugriffsfunktionen definieren. Die eigentlichen Daten, auf die darin zugegriffen wird, liegen vollständig im indirekten Format vor. Ist das System stärker direkt ausgeprägt, können die Attributwerte auch im Interfaceobjekt selbst abgelegt werden. Das Metaformat wird dann nur für die Anfrageauswertung und den internen Datenbankzugriff verwendet. Die Grenze zwischen diesen Datenformaten ist also nicht eindeutig festgelegt.

Dieser Ansatz vereinigt die beiden Vorteile der direkten und indirekten Abbildung: Natürliche Einbettung in die Programmiersprache bei generischer Typunterstützung zur Laufzeit. Gleichzeitig muß man aber auch Nachteile beider Verfahren in Kauf nehmen: man ist weiterhin vom Compiler abhängig, ohne jedoch dessen Effizienzgewinn voll ausnutzen zu können.

Das nachfolgende Beispiel erläutert die drei beschriebenen Abbildungsverfahren an einem einfachen Datenbankobjekt. Die Grafikelemente in Bild 10 folgen dabei der Designmethodologie OMT von Rumbaugh et al. ([RBP*91]).

Beispiel 4.2:

In Bild 10 sind die drei Abbildungsverfahren an einem Beispiel beschrieben. Oben im Bild findet man die Klassendefinition im konzeptuellen Schema. Hier wird eine Klasse namens Publication mit drei Attributen definiert: einer Identifikation, dem Titel und dem Preis.

Bei der direkten Abbildung wird hierfür ebenfalls eine C++-Klasse Publication erzeugt, die genau die angegebenen Attribute als Datenfelder enthält. Rechts neben der Klassendefinition findet man eine Beispielinstanz. Ein Implementierungsobjekt vom Typ Publication entspricht also genau einem Datenbankobjekt und trägt selbst die gesamte Information. Die verwendeten C++-Klassen sind für jedes Schema verschieden.

Bei der indirekten Abbildung wird dagegen eine Reihe vordefinierter Klassen verwendet. Jede Instanz der C++-Klasse Object stellt dabei ein Datenbankobjekt dar, unabhängig von der Datenbankklasse, zu der es gehört. Da jede Datenbankklasse anders aussieht, können die Attributwerte folglich nicht direkt als Datenfelder im Objekt gespeichert werden. Statt dessen verwaltet jedes Objekt eine Liste von Attributobjekten. Jedes dieser Attributobjekte enthält eine Beschreibung des repräsentierten Attributs sowie den zugehörigen Attributwert. Wie man auf der rechten Seite sieht, ist aus einem Datenbankobjekt nun ein Baum aus vier Implementierungsobjekten geworden.

Das am Ende skizzierte Mischverfahren definiert nun zusätzlich eine Wrapperklasse Publication, um auf die indirekte Repräsentation zuzugreifen. Aus der Sicht des Benutzers verhält sich jede Instanz dieser Klasse genau wie eine Publikation aus der direkten Abbildung; intern wird jedoch auf den generischen Strukturen gearbeitet. ■

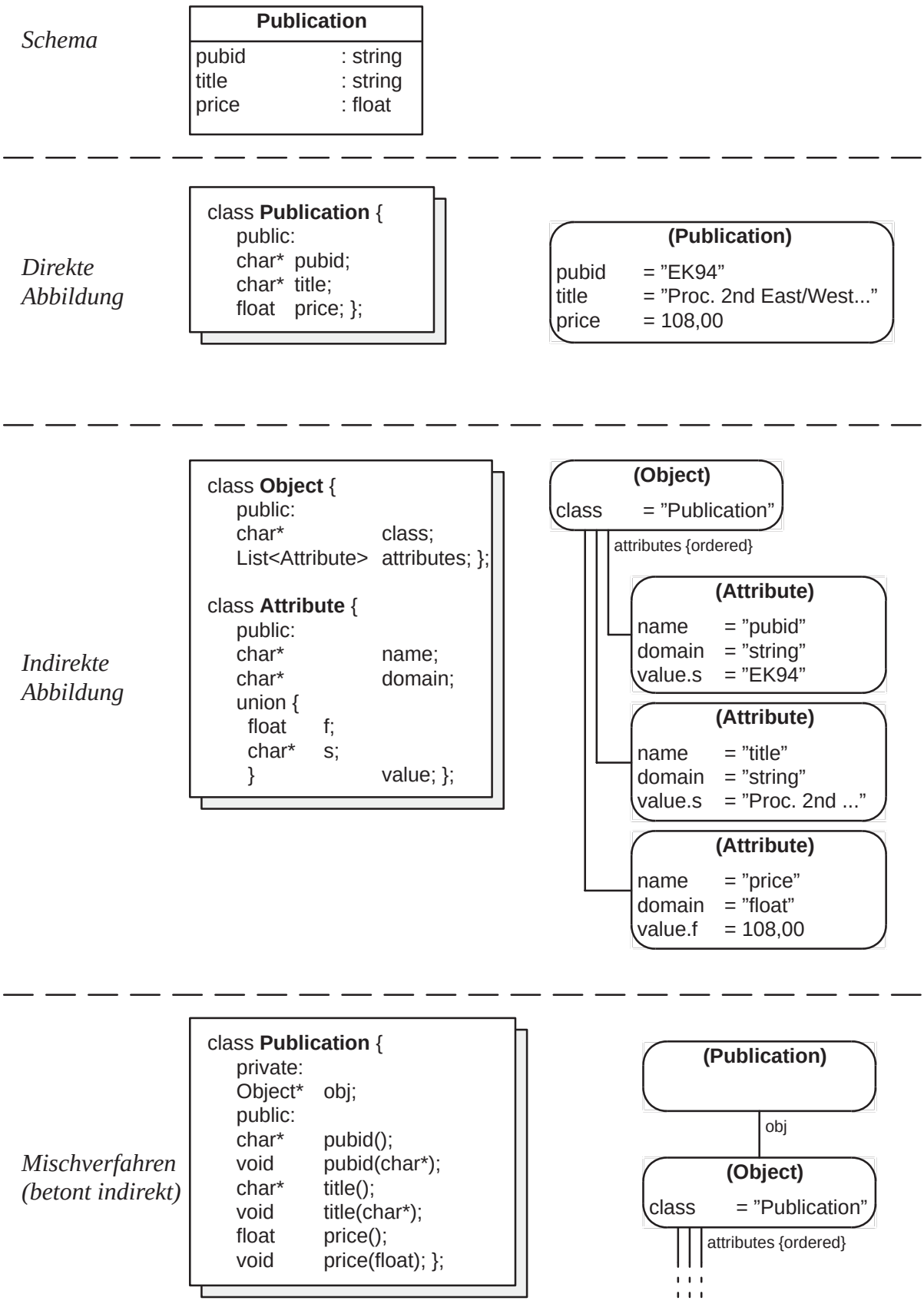


Bild 10: Abbildungsverfahren für C++-Systeme

4.2.3 Einschränkungen durch C++

Die vorgestellten Abbildungsverfahren sind unterschiedlich von den Eigenschaften der Sprache C++ abhängig. Bei der direkten Einbettung sind alle Datenbankobjekte äquivalente C++-Objekte und somit vollständig den Eigenschaften von C++ unterworfen. Bei der indirekten Einbettung kann hingegen eine in C++ nicht vorhandene Veränderbarkeit der Datenbankobjekte erreicht werden, was aber sowohl einen höheren Implementierungsaufwand für die Verwaltung der Objektnetze erfordert, als auch durch den interpretativ-navigatorischen Charakter zu einer geringeren Effizienz führt. In der Sprachbindung von ODMG–93 wird, wie auch in vielen anderen Systemen, die direkte Abbildung verwendet (siehe Kapitel 4.2.4). Folglich befasse ich mich nun speziell mit den Eigenschaften dieser Abbildungsform. Jedes Datenbankobjekt ist also durch ein C++-Implementierungsobjekt einer äquivalenten Klasse definiert. Hierdurch sind die Objekte direkt den Eigenschaften von C++ unterworfen. Dies umfaßt die eigentlichen Spracheigenschaften, die Kompilationsumgebung und die Eigenschaften des kompilierten Codes.

- Notwendigkeit eines Kompilationszyklus:
  Jede Änderung am Schema oder an der Objektstruktur erfordert einen erneuten Kompilationsschritt. Die jeweilige Anwendung muß angehalten und nach Übersetzung der Quelltexte wieder neu gebunden werden. Eine direkte Änderung zur Laufzeit ist nicht möglich.
- Hierarchische Klassenzugehörigkeit:
  Ein Objekt kann stets nur zu einer Klasse und ihren Superklassen gehören. Freie Klassenzugehörigkeit wird nicht unterstützt.
- Feste Klassenzugehörigkeit (Keine Migration):
  Die Klasse, zu der ein Objekt gehört, wird bei der Objekterzeugung festgelegt, sie kann später nicht mehr verändert werden.
- Feste Struktur (Keine Veränderbarkeit):
  Die Struktur einer Klasse wird zur Übersetzungszeit festgelegt. Die Struktur und das Verhalten eines Objekts können danach nicht mehr verändert werden.
- Fehlende Metainformation:
  Der Compiler überführt alle Objekte in Speicherblöcke, die in Folge alle Datenwerte sowie Zeiger auf Virtuelle Methodentabellen enthalten. Attributzugriffe werden in größen- und positionsabhängige Speicherblockoperationen umgesetzt. Das Objekt selbst enthält zur Laufzeit keinerlei Information mehr über seine eigene Klasse oder die in ihm enthaltenen Attribute, ihre Namen, Typen und Anfangsadressen. Mehrfachvererbung und virtuelle Basisklassen erzeugen zusätzlich objektinterne Zeiger und genestete Attributgruppen. Auch die Informationen über Methoden gehen verloren. Parameter werden auf den Aufrufstapel geschoben und dann wird durch einen Index zu einer Adresse in der Virtuellen Methodentabelle gesprungen. Es sind keine Informationen über den Methodennamen, Anzahl, Typ und Reihenfolge der Parameter oder den Rückgabewert verfügbar. Selbst die Virtuelle Methodentabelle ist für den Programmierer nicht zugreifbar und kann bei Mehrfachvererbung sogar auf mehrere Tabellen aufgeteilt sein. Durch die dem C++-Standard neu hinzugefügte *Run Time Type Identification (RTTI)* ([Str00]) kann diese Metainformation inzwischen teilweise abgefragt werden. Während der Entwicklungszeit des Systems waren jedoch noch keine Compiler verfügbar, die diese Erweiterung unterstützten.
- Keine generischen Zugriffe:
  Eine Folge der fehlenden Metainformation ist auch das Fehlen generischer Zugriffsmethoden. Ist der Name eines Attributs erst zur Laufzeit bekannt, kann nicht darauf zugegriffen werden, da Adresse und Typ des Attributs im Objekt nicht bestimmt werden können. Beim

Methodenaufruf kommt noch die notwendige Stapelmanipulation erschwerend hinzu. Während RTTI inzwischen zwar Metainformation verfügbar machen kann, enthält sie keine Unterstützung für generische Zugriffe.

- Keine dynamische Typerzeugung:
  Auch die Erzeugung neuer Typen zur Laufzeit ist nicht vorgesehen. Nur bei genauer Detailkenntnis über den verwendeten Compiler kann ein entsprechender Übersetzungsschritt im Laufzeitsystem nachempfunden werden. Die Datenbank Ontos verwendet ein solches Verfahren, bei dem C++-Objekte durch Speichermanipulationen zusammengesetzt werden. Dies funktioniert aber jeweils nur für einen ganz bestimmten Compiler auf genau einer Systemarchitektur und erfordert sehr genaue Informationen über dessen Interna.

All diese Einschränkungen müssen bei der Umsetzung des Sichtsystems beachtet und gegebenenfalls durch Implementierung von Hilfsklassen ausgeglichen werden.

4.2.4 Die C++-Sprachbindung in ODMG–93

Der in Kapitel 2.1 beschriebene Teil des Standards definiert ein generisches Datenmodell mit den zugehörigen Sprachkonstrukten zur Definition, Abfrage und Manipulation. Um ausgehend von dieser Spezifikation eine reale ODMG-Datenbank zu implementieren, gibt es prinzipiell zwei Möglichkeiten. Die erste Möglichkeit ist, das generische Modell unverändert als neue Schicht auf der verwendeten Programmiersprache aufzusetzen. Ein Schema wird in diesem Fall weiterhin in ODL spezifiziert, und um auf die Instanzen der Datenbank zuzugreifen, gibt es eigene OQL- und OML-Interpreter. Ein solcher interpretierender Ansatz ist etwa bei dem Nicht-ODMG-System VODAK verwendet worden und entspricht auch der Anbindung von Relationalen Datenbanken mit ihrem eigenen Typsystem und eingebetteten SQL-Kommandos ([Kla93], [KFM*96], vgl. Kapitel 6.6). Der Vorteil dieses Ansatzes ist die direkte Portierbarkeit des Schemas. Da sowohl das Schema als auch die Anwendung mit den generischen, standardisierten Sprachkonstrukten spezifiziert werden, kann die Spezifikation prinzipiell unverändert für jede ODMG-Datenbank übernommen werden, die diesem Ansatz folgt. Der Nachteil bei diesem Verfahren ist jedoch die Trennung des Typsystems und der Programmiersprache der Datenbank vom Typsystem und der Programmiersprache der Anwendung (*semantic gap* oder *impedance mismatch*). Es ist jedoch gerade ein Vorteil des objektorientierten Modells, daß diese Lücke überbrückt werden kann. Aus diesem Grunde folgt der ODMG-Standard auch dem anderen Ansatz, nämlich der nahtlosen Integration des Datenbanksystems in die vorhandene Programmierumgebung. Bei dieser Variante soll der Endanwender keinen Unterschied zwischen Datenbankzugriffen und der normalen Arbeit in seiner Programmiersprache erkennen können. Es werden also direkt die Möglichkeiten der jeweiligen Sprache ausgenutzt, und das generische Modell dient nur noch als semantische Basis, ohne aktiv benutzt zu werden. Es ist dies das ausdrückliche Ziel der Object Database Management Group:

> *“1. There is a single unified type system across the programming language and the database; individual instances of these common types can be persistent or transient.*
>
> *2. The programming language-specific binding for ODL/OML respects the syntax and semantics of the base programming language into which it is being inserted.*
>
> *3. The binding is structured as a small set of additions to the base programming language; it does not introduce sublanguage-specific constructions that duplicate functionality already present within the base language.*

4. Expressions in the OML compose freely with expressions from the base programming language and vice versa." [Cat94, S. 84]

Um diese Integration zu erreichen, werden im Standard bereits eine Reihe sogenannter *Sprachbindungen* (*language bindings*) definiert. Sie legen fest, wie das Datenmodell und die Definitions-, Manipulations- und Anfragesprachen in der jeweiligen Programmiersprache darzustellen sind. Bisher existieren nur Bindungen für C++ und Smalltalk, sowie neuerdings für Java. Laut ODMG befinden sich Bindungen für C, Pascal, CLOS und IDL bereits in der Planung. Im folgenden wird nur auf die C++-Sprachbindung eingegangen; die anderen Sprachbindungen finden sich in [Cat93-96] bzw. [CaB97-00]. Die Sprachbindung für C++ ist die wichtigste Sprachbindung und nimmt in ODMG–93 den größten Raum ein. Diese Bedeutung spiegelt sich auch in der Praxis wider: Fast immer, wenn von Standard-Konformität (*ODMG compliance*) gesprochen wird, ist hierbei die Übereinstimmung mit der C++-Bindung gemeint. In der Sprachbindung wird für alle vier Elemente des generischen Teils, also Datenmodell, Spezifikation (ODL), Manipulation (OML) und Abfrage (OQL), eine Abbildung in die C++-Umgebung definiert.

C++-Datenmodell

Da das Typsystem von ODMG sehr stark an C++ orientiert ist, ist die Abbildung auf das C++-Datenmodell in weiten Teilen offensichtlich. Die atomaren Literaltypen werden größtenteils auf die entsprechenden C++-Datentypen abgebildet; für die Collection-Typen und variablen Arrays werden Template-Klassen definiert; Aufzählungen, Strukturen und Arrays fester Länge werden unverändert nach C++ übernommen. Atomare Objekttypen werden in C++ als Klassen definiert, wobei Attribute und Relationships zu Datenfeldern des Objekts werden. Zur Darstellung von Relationships wird ein spezieller Referenzdatentyp `Ref<T>` definiert.[18] Unidirektionale Relationships zählen zu den Attributen und werden einfach durch ein `Ref<>`-Element repräsentiert. Für bidirektionale Relationships werden bestimmte Relationship-Klassen implementiert, die sich neben der Navigation auch um die Zusicherung referenzieller Integrität kümmern (siehe Kapitel 4.4). Die im generischen Teil definierten Lese- und Schreiboperationen auf Attributen (`get_value` und `set_value`) sind in der C++-Sprachbindung leider aufgegeben worden ([Cat94, S. 90]). Diese Funktionalität kann aber dennoch nachgebildet werden, indem man die Attribute auf private Datenfelder abbildet und zusätzliche öffentliche Lese- und Schreibmethoden generiert. Dieser Ansatz ist zum Beispiel bei der ODMG-Library für Ontos verwendet worden ([GOP96]) und ist auch bei der Definition von Sichten notwendig. Eine genaue Spezifikation der verwendeten Sprachbindung findet sich in Anhang H.1.[19]

C++-ODL

Das zweite Element in der C++-Bindung ist die Spezifikationssprache C++-ODL. Ihre Rolle ist im Standard nicht genau festgelegt; sie ist eine Art Zwischenformat zwischen dem generischen ODL und reinem C++ und widerspricht genaugenommen in ihren Eigenschaften den Zielsetzungen des Standards (siehe Anhang G). Für das Sichtsystem wurde daher auf diese Zwischensprache verzichtet. Die Sprachbindung wird mit Hilfe eines Präprozessors direkt aus reinem ODL generiert (siehe Abschnitt 4.6.1).

---

18. In Release 1.1 des Standards ist die genaue Implementierung einer Relationship noch offen; sie soll den Referenztyp verwenden, und die referenzielle Integrität wird vom Datenbanksystem zugesichert. In Release 1.2 werden zu diesem Zweck normierte genestete Relationshipklassen definiert (siehe Abschnitt 6.2.1).

19. Im Rahmen von ODMG 1.2 ist die Sprachbindung in weiten Teilen geändert worden. Statt der Standard-Datentypen wird jetzt eine ODMG-spezifische Typbibliothek verwendet (siehe Abschnitt 6.2.1).

C++-OML

Obwohl ODMG–93 keine ausgesprochene Manipulationssprache definiert, sind die Möglichkeiten zur Objekt- und Datenmanipulation dennoch im Standard festgelegt. Der eine Teil der Operationen ergibt sich aus C++ selbst. Da alle ODMG-Datentypen auf C++-Datentypen abgebildet werden, stehen automatisch alle C++-Operationen zur Verfügung. Neben den arithmetischen Basisoperatoren und den Funktionen der Standardbibliotheken gehört dazu ebenfalls die Möglichkeit, neue Funktionen zu definieren. Der zweite Teil von C++-OML ergibt sich aus der Definition der ODMG-Typen, die über die C++-Basistypen hinausgehen. Zur Darstellung von Mengen, Arrays, Referenzen, usw. werden neue C++-Klassen definiert. Diese Klassen gehören mit zum Standard und legen die Operationen fest, die auf den jeweiligen Objekten ausgeführt werden können. Die Klassendefinition für Mengen verlangt zum Beispiel Funktionen zum Einfügen und Entfernen von Objekten aus der Menge. Neben den Literalklassen werden weiterhin auch Objektklassen vordefiniert. Hierzu gehört die persistente Basisklasse Pobject, sowie die Klassen Database und Transaction. Die Collection-Klassen werden über Templates definiert, deren Interfaces ebenfalls vordefiniert sind. Leider stimmen die im Standard definierten Datentypen der Sprachbindungen und ihre Interfaces nicht mit den Typen und Funktionen des generischen Teils überein (siehe Anhang G). Diese verschiedenen Definitionen ergeben eine ODMG-Klassenbibliothek, die zur Laufzeit von der Anwendung verwendet wird.

C++-OQL

Die Elemente von C++-OML erlauben bereits, Daten aus der Datenbank zu lesen und zu verarbeiten. Neben diesem prozeduralen Ansatz gibt es in der C++-Sprachbindung zwei Mechanismen zur Auswertung deklarativer Anfragen. Eine Möglichkeit ist die Selektion auf einer Collection. Die vordefinierte Funktion Collection::query() erhält als Argument eine Zeichenkette, die einen logischen OQL-Ausdruck enthält. Die Ausführung von 'col1.query(col2,"pred")' ist gleichbedeutend mit der OQL-Anfrage 'col2 = **select** this **from** this **in** col1 **where** pred'. Das Schlüsselwort this bezeichnet dabei das jeweils aktuelle Element. Die zweite Möglichkeit ist die Auswertung beliebiger OQL-Anfragen. Die Template-Funktion oql(res,query,para) wertet den übergebenen Anfragetext query aus und weist das Ergebnis der Variablen res zu. Innerhalb der Anfrage kann dabei auf die Parameterliste para zugegriffen werden. Wie eine solche Anfrage angesichts der fehlenden generischen Zugriffsmethoden in kompiliertem C++-Code allerdings ausgeführt werden soll, ist im Standard nicht erwähnt. Der in meinem System verwendete Lösungsansatz ist in Abschnitt 4.5 zu finden.[20]

Anmerkungen zur Sprachbindung

Genau wie beim generischen Teil des Standards gibt es auch bei der Sprachbindung Probleme und Ungenauigkeiten. Die Sprachbindung selbst ist nicht konkret definiert, sondern nur anhand von Beispielen erläutert. Diese Beispiele widersprechen sich wiederum in einzelnen Fällen. Bei der hier verwendeten Abbildung handelt es sich also nur um eine mögliche Interpretation. Andere C++-Sprachbindungen können davon abweichen. Eine detaillierte Liste von Problemen ist in Anhang G aufgeführt. Die dort geschilderten Probleme haben Einfluß auf den Entwurf und die Umsetzung des Sichtsystems. Manche Ideen waren hierdurch nicht umsetzbar.

4.2.5 Sichtspezifische Ergänzungen der Sprachbindung

Für eine Darstellung der Sichtklassen in der C++-Sprachbindung muß die vorgestellte Standard-Sprachbindung an die sichtspezifischen Elemente angepaßt werden. Die Rückverweise werden

20. Auch hier gibt es wieder deutliche Änderungen in Release 1.2 (siehe Abschnitt 6.2.1).

als geschützte Datenfelder berücksichtigt. Für jedes Attribut wird eine Lese- und eine Schreibmethode definiert, die durch Navigation über die Rückverweise den Attributwert berechnet bzw. einen geänderten Wert in die Originalobjekte zurückschreibt. Zur Darstellung von Relationships sind zusätzliche Klassen notwendig, die sich um Traversierung und Konsistenzerhaltung kümmern. Weiterhin muß bei der Abbildung von Datenbankklassen auf Implementierungsklassen die komplexe Interaktion mit den Laufzeitkomponenten berücksichtigt werden.

4.2.5.1 Datentypen

Die Sprachbindung des Sichtsystems folgt der im vorigen Abschnitt gegebenen Sprachbindung von ODMG–93, bei der alle Datentypen und Datenbankklassen direkt in C++ eingebettet werden. Die Abbildung von Datentypen ist in Anhang H.1, Tabelle 8 vollständig definiert. Sie enthält bereits die von mir verwendete Interpretation des Standards einschließlich selbst definierter Elemente für Zweifelsfälle.

4.2.5.2 Implementierungsklassen

Bei der Abbildung von Klassen folge ich grundsätzlich der Original-Sprachbindung (siehe Anhang H.1, Tabelle 9), bilde jedoch Datenfelder nicht auf öffentliche Attribute ab, sondern lege private Attribute an, auf die mit Lese- und Schreibmethoden zugegriffen wird. Dies hat zwei Vorteile: Erstens kann die im Standard geforderte Ausführung von mark_modified nach einer Wertänderung im Schreibzugriff versteckt werden; sie muß nicht mehr explizit vom Anwendungsprogramm aufgerufen werden, was Inkonsistenzen verhindert. Zweitens kann die Berechnung virtueller Datenfelder direkt durch die Zugriffsmethoden erfolgen. Ein asynchrones Berechnen und Zurückschreiben der Attributwerte wird dadurch vermieden. Bei der Klassenabbildung muß man zusätzlich noch zwischen normalen, Import- und Konvertierungsklassen unterscheiden.

Normale Klassen

Normale Klassen ohne Import- oder Konvertierungsfunktionalität werden wie beschrieben abgebildet. Die Implementierungsklasse ist Subklasse der in der ODL-Spezifikation angegebenen Superklassen in unveränderter Reihenfolge. Ist keine Superklasse angegeben, wird die neue Klasse von Pobject abgeleitet. Für jedes Attribut werden ein privates Datenfeld und zwei Zugriffsfunktionen angelegt. Die Lesemethode liefert den im Datenfeld abgelegten Wert zurück, die Schreibmethode löscht den alten Wert, weist den neuen zu und markiert das Objekt als verändert. Für Relationships werden spezielle Implementierungsklassen verwendet, die die referenzielle Integrität sicherstellen (siehe Abschnitt 4.4).

Beispiel 4.3:

Gegeben sei die folgende ODL-Spezifikation einer Klasse:

```
interface Author
    ( extent authors key (lastname, firstname) ) : persistent
    {
    attribute     string              lastname;
    attribute     string              firstname;
    relationship  set<Publication>    publications inverse Publication::authors;
                  long                pubcount();
    };
```

Die hieraus generierte Sprachbindung führt zu der folgenden C++-Klassendeklaration:

```
class Author : public Pobject
    {
    protected:   char* priv_lastname;                          // attributes
                 char* priv_firstname;
    public:      SetRel<Publication> publications;             // relationships
    public:      virtual const char* lastname () const;        // public access methods
                 virtual void lastname (const char* newval);
                 virtual const char* firstname () const;
                 virtual void firstname (const char* newval);
    public:      virtual long pubcount();                      // methods
    };
```

Hinzu kommen noch Konstruktoren, Destruktoren und weitere Hilfsfunktionen. ■

Importklassen

Jede Importklasse ist Subklasse der in der ODL-Spezifikation angegebenen Superklassen in unveränderter Reihenfolge. Ist keine Superklasse angegeben, wird die neue Klasse von ImportClass abgeleitet. Alle Superklassen müssen ebenfalls Importklassen sein und von der gleichen externen Datenbank importieren. Die zugehörigen externen Klassen müssen in der gleichen Vererbungsbeziehung stehen (Konsistenzbedingung von Seite 47).

Um Netzwerkzugriffe zu vermindern, werden Datenfeldzugriffe auf Importobjekte nicht direkt an die externen Objekte weitergeleitet, sondern auf lokalen Kopien der Datenfelder ausgeführt. Alle Datenfelder von Importklassen sind also Cache-Datenfelder. Genau wie für normale Klassen wird auch für Importklassen für jedes Attribut ein privates Datenfeld angelegt, auf das mit zwei Zugriffsfunktionen zugegriffen wird. Relationships werden wiederum mit speziellen Implementierungsklassen realisiert, die sich aber von denen normaler Klassen unterscheiden. Die Datenfelder werden bei der Objektaktivierung vom Objektmanager initialisiert. Das Zurückschreiben etwaiger Änderungen erfolgt bei der Deaktivierung automatisch vom Objektmanager. Für den Rückverweis ist keine Implementierung notwendig, da das Datenfeld priv_orig und die beiden Zugriffsmethoden von der Basisklasse ImportClass geerbt werden. Die Methoden der Importklassen können automatisch generiert werden, da sie stets die zugehörigen Methoden des externen Originalobjekts aufrufen müssen. Hierbei werden der Methodenname und die Parameter an die externe Datenbank übertragen und das Ergebnis von dort geholt. Die hierbei notwendigen Synchronisationsaufgaben werden in Kapitel 5 beschrieben.

Bei meinem Ansatz wird stets der komplette Objektzustand gespeichert. Alternativ ist es auch möglich, bei der Aktivierung die Datenfelder als nicht-initialisiert zu markieren und die Daten attributweise beim ersten Zugriff über die Lesemethode nachzuladen ([Emp97]). Dieser Ansatz ist vorzuziehen, wenn hauptsächlich nur wenige Attribute eines ansonsten großen Objekts benötigt werden. Für vorwiegend kleine Objekte ist die Komplettübertragung des Zustands in einer einzigen Aktion effizienter, und die Gültigkeitsüberprüfung bei jedem Lesezugriff kann entfallen. Eine mögliche Optimierung ist die Initialisierung ausgewählter Attribute bei der Aktivierung kombiniert mit dem Nachladen der restlichen Attribute. Dies könnte etwa durch eine Erweiterung der Präprozessor-Konfigurationsdatei (Kapitel 4.6.1) erfolgen.

Beispiel 4.4:

Gegeben sei die folgende ODL-Spezifikation einer Importklasse:

```
interface PUBDB_Author : ImportClass
    (   extent pubdb_authors imports Author from PUBDB::authors;
        key (lastname, firstname) ) : virtual
    {
    attribute       string                          lastname;
    attribute       string                          firstname;
    relationship    set<PUBDB_Publication>          publications inverse PUBDB_Publication::authors;
                    long                            pubcount ();
    };
```

Die hieraus generierte Sprachbindung führt zu der folgenden C++-Klassendeklaration:

```
class PUBDB_Author : public ImportClass
    {
    protected:  char* priv_lastname;                        // attributes
                char* priv_firstname;
    public:     SetRel<PUBDB_Publication> publications;     // relationships
    public:     virtual const char* lastname () const;      // public access methods
                virtual void lastname (const char* newval);
                virtual const char* firstname () const;
                virtual void firstname (const char* newval);
    public:     virtual long pubcount ();                   // methods
    };
```

Abgesehen von der Superklasse ImportClass ist der Klassenaufbau identisch mit dem normaler Klassen. Unterschiede ergeben sich erst im Methodencode und in den Hilfsfunktionen. ■

Konvertierungsklassen

Auch bei Konvertierungsklassen wird die Liste der Superklassen übernommen; fehlt sie, so wird von DerivedClass abgeleitet. Abgesehen von DerivedClass müssen alle Superklassen einer Konvertierungsklasse ebenfalls Konvertierungsklassen sein. Normale Attribute und Relationships werden wie bei normalen Klassen implementiert. Für virtuelle Attribute werden zwar Lese- und Schreibmethoden aber keine Datenfelder angelegt. Die Lesemethode berechnet den Attributwert direkt aus den Basisobjekten des Konvertierungsobjekts und die Schreibmethode führt direkt die notwendigen Manipulationen auf den Basisobjekten aus. Im Gegensatz zu Importklassen muß für jede Konvertierungsklasse, die direkt von DerivedClass abgeleitet wird, ein eigenes orig-Attribut angelegt werden, da sich seine Struktur für die verschiedenen Klassen unterscheidet. Für Relationships wird wiederum eine spezifische Hilfsklasse verwendet.

Beispiel 4.5:

Gegeben sei die folgende ODL-Spezifikation einer Konvertierungsklasse:

```
typedef struct { ADRDB_Person person; PUBDB_Author author; } Person_OrigType;

interface Person : DerivedClass
    (   extent persons is   select Person ( orig: struct( person:p, author:a ) )
                            from p in adrdb_persons, a in pubdb_authors
                            where p.lname=a.lastname and p.fname=a.firstname
                        union
                            select Person( orig: struct( person:p, author:nil ) )
                            from p in adrdb_persons
                            where not exists a in pubdb_authors:
                                (p.lname=a.lastname and p.fname=a.firstname);
```

```
    key (lname,fname) ) : virtual
{
private   attribute     Person_OrigType  orig;
          attribute     string           lname      is this.orig.person.lname;
          attribute     string           fname      is this.orig.person.fname;
          attribute     string           birthdate  is this.orig.person.birthdate;
          relationship  Address          address    inverse Address::residents is
              element(select a from a in addresses
              where a.orig.address = this.orig.person.address);
};
```

Auffällig an der Klassenspezifikation ist die relativ komplexe Extentklausel. Zur Bestimmung der Personen würde eigentlich eine Selektion über adrdb_persons ausreichen, um aber eine Subklasse Authors zu unterstützen, muß die entsprechende Referenz bereits in der Person-Klasse zur Verfügung gestellt werden. Damit auch die Personen übernommen werden, die keine Autoren sind, muß man in der Extentklausel einen Outer Join verwenden, der für die korrespondenzfreien Personen den Autor auf nil setzt (siehe Kapitel 3.7.2). Die hieraus generierte Sprachbindung führt zu der folgenden C++-Klassendeklaration:

```
typedef struct _Person_OrigType
   { Ref<ADRDB_Person> person; Ref<PUBDB_Author> author; } Person_OrigType;

class Person : public DerivedClass
   {
   protected:   Person_OrigType priv_orig;                    // attributes
   public:      SimpleRel<Address> address;                   // relationships
   public:      virtual const char* lname () const;           // public access methods
                virtual void lname (const char* newval);
                virtual const char* fname () const;
                virtual void fname (const char* newval);
                virtual const char* birthdate () const;
                virtual void birthdate (const char* newval);
   protected:   virtual const Person_OrigType& orig () const;  // protected access methods
                virtual void orig (const Person_OrigType& newval);
   };
```

In der Sprachbindung wird nur für orig ein Datenfeld angelegt, alle anderen Attribute werden ausschließlich über Zugriffsfunktionen dargestellt. ■

4.2.5.3 Vordefinierte Klassen

Damit die erzeugten Klassen der Sprachbindung arbeiten können, muß eine Reihe weiterer, schemaunabhängiger C++-Klassen vordefiniert sein (siehe Anhang H). Am wichtigsten für die Sprachbindung sind dabei die drei Basisklassen für alle virtuellen Klassen (siehe Bild 11). Die vollständigen Klassendeklarationen sind in Anhang H.7 zu finden.

- Pobject oder Persistent_Object ist gemäß ODMG–93 die Basisklasse aller Datenbankklassen. Sie definiert die im Standard geforderten Funktionen sowie Konstruktoren, Zugriffsfunktionen und Hilfsmethoden für die Objektverwaltung. Alle normalen Klassen einer Sicht werden von Pobject abgeleitet.
- ImportClass ist eine Subklasse von Pobject. Sie ergänzt Pobject um die gemeinsame Grundfunktionalität aller Importklassen. Da alle Importklassen dieselbe Struktur für ihr orig-Attribut verwenden, nämlich eine einzelne Referenz auf ein externes Objekt in Form einer GOID, kann das orig-Attribut bereits in ImportClass definiert und an alle Importklassen vererbt werden. Sämtliche Importklassen werden von ImportClass abgeleitet. Auf die GOID wird im folgenden Kapitel über die Objektverwaltung näher eingegangen.

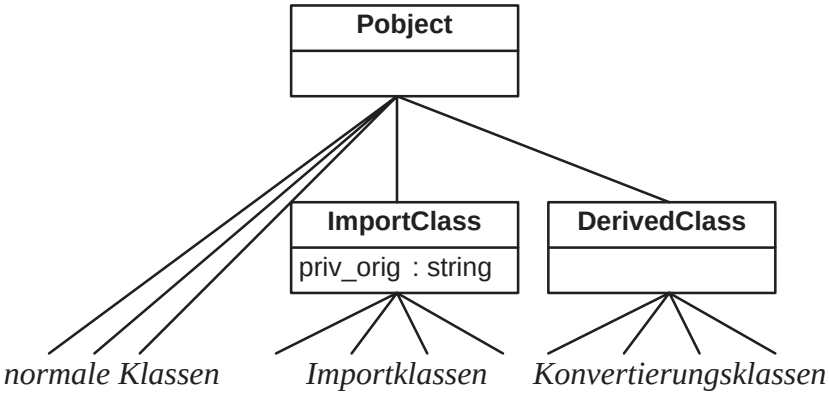


Bild 11: Klassenhierarchie der Sprachbindung

- DerivedClass ist ebenfalls eine Subklasse von Pobject und bildet die Basisklasse für alle Konvertierungsklassen. Sie enthält kein orig-Attribut, da die Struktur der Rückverweise für jede Konvertierungsklasse anders aussehen kann. Statt dessen müssen zwei Funktionen von jeder Subklasse zur Verfügung gestellt werden. Mit priv_hasKey wird überprüft, ob der Rückverweis eines Objekts mit dem Funktionsargument übereinstimmt, und priv_destroy muß durch geeignete Manipulationen der Basisobjekte das virtuelle Objekt aus der Datenbank entfernen (siehe Seite 90, Objektlöschung).

Das bisher vorgestellte Grundgerüst der Sprachbindung muß noch um weitere Methoden für das Zusammenspiel mit der Objektverwaltung und der Anfragebearbeitung ergänzt werden. Diese werden jeweils in den folgenden Abschnitten beschrieben.

4.3 Die Objektverwaltung

Das zweite Modul der Sichtimplementierung ist die Objektverwaltung. Sie dient der Erzeugung, Aktivierung, Deaktivierung und Löschung von Objekten und hat im wesentlichen die folgenden drei Hauptaufgaben:

- Automatischer Zugriff auf die Netzwerkkomponente zur Implementierung von Importobjekten
- Vermeidung unnötiger Objektinstanziierungen, um die Netzlast zu minimieren und Sichtzugriffe effizient zu behandeln
- Zusicherung der Identitätserhaltung bei partiellen und sich überschneidenden Objektzugriffen.

Diese Funktionen werden mit Hilfe eines Referenzmechanismus realisiert, der ein selektives Aktivieren und Deaktivieren von Objekten ermöglicht und im folgenden beschrieben wird:

Der Referenzmechanismus

Will eine Anwendung auf die Datenbank zugreifen, so müssen die entsprechenden Implementierungsobjekte im Hauptspeicher angelegt werden. Diese Aktivierung kann entweder komplett zu Beginn einer Transaktion geschehen oder je nach Bedarf objektweise ausgeführt werden. Lese- und Schreibzugriffe auf ein Implementierungsobjekt werden durch dieses Objekt auf die entsprechenden Datenbankoperationen abgebildet. Spätestens beim Ende der Transaktion werden die geänderten Daten aus dem Hauptspeicher in den Sekundärspeicher übertragen und stehen dann persistent zur Verfügung. Die Implementierungsobjekte können dann deaktiviert werden.

Der Prozeß des Aktivierens und Deaktivierens muß hierbei vom Generieren und Löschen persistenter Objekte unterschieden werden. Sowohl beim Aktivieren als auch beim Generieren eines Datenbankobjekts wird im Hauptspeicher ein neues Implementierungsobjekt angelegt. Der Unterschied liegt im Zusammenspiel mit der Datenbank. Im Falle der Aktivierung erfolgt ein Lesezugriff auf die Datenbank. Es wird ein Objekt, das bereits existiert aber im Augenblick nicht im Hauptspeicher vorliegt, in den Speicher zurückkopiert. Die Datenbank wird hierbei nicht verändert. Im Gegensatz dazu wird beim Generieren ein neues Objekt in die Datenbank eingetragen — es kommt also zu einem Schreibzugriff. Beim Deaktivieren und Löschen gilt das Umgekehrte: Deaktivierung entfernt nur aus dem Speicher, Löschen entfernt das Objekt zusätzlich aus der Datenbank.

Definition 4.3: Objekterzeugung, Objektlöschung

Objekterzeugung ist das Hinzufügen eines neuen Datenbankobjekts in die Datenbank. Das Entfernen eines Datenbankobjekts aus der Datenbank wird als *Löschung* bezeichnet. Beide Operationen stellen Updates dar.

Bei virtuellen Objekten ist zu beachten, daß man diese nicht direkt erzeugen kann, sondern durch entsprechende Operationen auf den Basisklassen dafür sorgen muß, daß durch Auswertung der Extent-Klausel ein neues Objekt in der virtuellen Klasse erscheint. Auch die Löschung eines virtuellen Objekts erfolgt indirekt durch Löschungen auf den Basisklassen.

Definition 4.4: Objektaktivierung, Objektdeaktivierung

Bei der *Objektaktivierung* wird ein Objekt vom persistenten Sekundärspeicher in den Arbeitsspeicher gebracht. Es werden also die zu einem existierenden Datenbankobjekt gehörenden Implementierungsobjekte angelegt. Unter *Deaktivierung* versteht man das Entfernen dieser Implementierungsobjekte aus dem Arbeitsspeicher, nachdem gegebenenfalls etwaige Updates auf den Sekundärspeicher geschrieben worden sind. Aktivierung und Deaktivierung stellen also keine Datenbank-Updates dar.[21]

Für das Aktivieren und Deaktivieren der Implementierungsobjekte gibt es derzeit zwei Verfahren: Fehlersignalmaskierung und Referenzobjekte. Die Fehlersignalmaskierung wird zum Beispiel in ObjectStore ([ODI95]) angewandt. Hier wird der Datenbankinhalt direkt in den adressierbaren Hauptspeicher eingeblendet. Trifft man beim Dereferenzieren eines Zeigers auf nicht-initialisierte Speicherbereiche (*page fault*), so wird das erzeugte Fehlersignal abgefangen und die zugehörige Datenbankseite an die entsprechende Stelle im Speicher gebracht. Das Zurückschreiben in die Datenbank erfolgt dann entweder direkt bei jeder Änderung auf den Objekten (*write through*) oder geschlossen bei systembedingter Freigabe der Speicherseite (*write cache*). Eine feingranulare Objektaktivierung kann man hingegen durch Verwendung sogenannter Referenzobjekte erreichen:

Definition 4.5: Referenz, Smart Pointer

Eine *Referenz* (*Referenzobjekt, smart pointer*) ist eine Art Zeiger, der über zusätzliches Wissen über das referenzierte Objekt verfügt.

Durch dieses zusätzliche Wissen können Referenzen die Objekte bei Bedarf automatisch aktivieren, sie können Referenzzähler zur automatischen Speicherfreigabe beinhalten (*garbage collec-*

21. Das Schreiben der Änderungen vor einer Deaktivierung ist zwar ein Update, dieses ist aber nicht von der Deaktivierung verursacht, sondern von den ursprünglichen Operationen, die die Implementierungsobjekte verändert haben. Die Deaktivierung bestimmt nur den spätesten Zeitpunkt für die Weiterleitung dieser Änderungen an den Sekundärspeicher.

tion), und sie können zur Aufrechterhaltung referenzieller Integrität herangezogen werden. Referenzen werden zum Beispiel in Ontos (OC_Reference [ONT94a]) und in ODMG–93 (Ref<>) verwendet. Sie bilden daher auch einen wichtigen Teil des Sichtsystems.

Definition 4.6: Late Instantiation, Bedarfsaktivierung

Bei der *Bedarfsaktivierung* oder *verzögerten Aktivierung* (*late instantiation*) wird ein Objekt erst dann aktiviert, wenn zum ersten Mal darauf zugegriffen wird. Im Gegensatz dazu wird bei *früher Aktivierung* ein Objekt bereits dann aktiviert, wenn es für das System sichtbar wird, also etwa beim Zugriff auf den Extent einer Klasse.

Bedarfsaktivierung verhindert das unnötige Aktivieren von Objekten, auf die später nicht zugegriffen wird, und sie erleichtert das Aktivieren von Objekten mit transitiven und zyklischen Abhängigkeiten. Andererseits führt sie, speziell in einem Sichtsystem mit berechneter Objektidentität, zu erhöhtem Verwaltungsaufwand. In meinem Sichtsystem wird ein spezieller Referenzmechanismus verwendet, um Objekte einzeln zu aktivieren und zu deaktivieren. Anstatt sämtliche Sichtklassen vollständig instanziiert vorliegen zu haben, werden nur die jeweils notwendigen Objekte angelegt. Um ein Objekt automatisch aktivieren zu können, braucht die Referenz geeignete Informationen über das von ihr referenzierte Objekt. Diese Objektinformationen werden von einer Referenz-Verwaltung (*ReferenceHandler*) organisiert. Die Referenzen selbst bekommen eine Nummer zugeordnet, mit der sie über die Verwaltung auf die Objektinformationen und schließlich auf das Objekt selbst zugreifen können. Für die Referenzen verwende ich den im Standard vorgesehenen Ref-Datentyp, der als ungetypte Referenz Ref_Any implementiert ist und in Form von C++-Templates getypte Referenzen Ref<T> zur Verfügung stellt (siehe Anhang H.2).

Definition 4.7: Referenznummer, RID

Eine *Referenznummer* (*Referenz-Id*, *RID*) ist ein von einer Referenz-Verwaltung vergebener ganzzahliger, nichtnegativer Wert $r \in RID=\{0,..,RID_{max}\}$. Sie bestimmt eineindeutig ein von der Referenz-Verwaltung adressierbares Objekt, das heißt, jedes Objekt besitzt genaue eine RID, und keine zwei Objekte besitzen die gleiche RID. Die RID wird zur Laufzeit vergeben und ist jeweils bis zum Ende einer Sitzung gültig. Die RID 0 (Null) ist für Referenzen ohne Ziel reserviert (*Null-Referenz*, *Nil*), sie verweist auf kein Objekt.

Die Speicherung der Objektinformationen in einem Verwaltungsobjekt statt in der Referenz selbst hat den Nachteil, daß bei jedem Zugriff über eine Referenz eine zusätzliche Indirektion notwendig ist. Dem gegenüber steht eine Reihe von Vorteilen:

- Die Referenzen benötigen nur wenig Speicherplatz.
- Die Objektinformation wird nicht dupliziert.
- Referenzvergleiche sind reine numerische Vergleiche und damit sehr schnell.
- Der Referenztyp ist unabhängig von der zur Objektaktivierung notwendigen Information.
- Objekte können aktiviert und deaktiviert werden, ohne Referenzen verändern zu müssen.

Auf Grund der unterschiedlichen Anforderungen von normalen, Import- und Konvertierungsobjekten werden im System verschiedene Verwaltungsobjekte verwendet. Diese werden in einer globalen Liste geführt und über einen Index ausgewählt. Dieser Index muß ebenfalls in den Referenzen gespeichert werden, um das zuständige Verwaltungsobjekt wiederzufinden. Bei Bedarf

können weitere Verwaltungsobjekte in der Liste registriert werden, um neue Aktivierungsmechanismen hinzuzufügen.[22]

Definition 4.8: Verwaltungsindex, Erweiterte Referenznummer, XRID

Eine *erweiterte Referenznummer* xrid∈XRID ist ein Paar (vid,rid), bestehend aus einem *Verwaltungsindex* vid∈VID={0,..,VID_{max}} und einer Referenznummer rid∈RID. Die Zuordnung zwischen allen adressierbaren Objekten und ihren XRIDs ist eineindeutig. Alle Referenzen auf ein Objekt verwenden die gleiche XRID. Alle Null-Referenzen gelten als gleich: $(v_1,r_1)=(v_2,r_2) \Leftrightarrow r_1=r_2 \wedge (v_1=v_2 \vee r_1=0)$.

In meinem System ist $VID_{max}=2^3-1=7$ und $RID_{max}=2^{29}-1=536870911$. Eine vollständige erweiterte Referenznummer ist also in einem 32bit-Wert darstellbar. Im Gegensatz zur Originaldefinition 4.7 sind RIDs jetzt nur noch innerhalb eines Verwaltungsindex eindeutig.

In meinem Sichtsystem kann man die folgenden vier Verwaltungsobjekte unterscheiden (siehe Anhang H.2):

a) Die *Hauptspeicher-Objektverwaltung* (MemRefHdlr) dient dazu, beliebige, von Pobject abgeleitete Objekte in das Referenzsystem zu integrieren. Sie kann nur Objekte verwalten, die sich bereits im Speicher befinden. Objektaktivierung wird nicht unterstützt, und die Verwaltung von Schlüsselwerten entfällt.

b) Die *Import-Objektverwaltung* (ImpRefHdlr) ist für alle Importobjekte zuständig. Als Schlüsselwert wird ein globaler Objektidentifier (GOID) verwendet, der jedes externe Objekt eindeutig identifiziert. Bei Aktivierung und Deaktivierung wird mit Hilfe der GOID auf die externe Datenbank zugegriffen, um die notwendigen Daten zu übertragen. Derzeit wird nur ein einzelnes Verwaltungsobjekt für sämtliche Importobjekte verwendet; bei unterschiedlichen Aktivierungsmechanismen oder Schlüsselformaten können auch verschiedene Import-Objektverwaltungen eingesetzt werden.

c) Die *Konvertierungs-Objektverwaltung* (DerivRefHdlr) verwaltet alle Konvertierungsobjekte. Da jede Konvertierungsklasse einen anderen Typ für ihre Schlüsselinformation verwenden kann, gestaltet sich die Zuordnung zwischen Schlüsselwerten und Objekten schwieriger als im Importfall. Als Schlüsselwert muß nun der Klassenname zusammen mit der in dieser Klasse verwendeten Orig-Information verwendet werden. Will man wie im Importfall die Schlüsselinformationen direkt in der Objektverwaltung halten, so müssen alle Orig-Typen von einer gemeinsamen Grundstruktur abgeleitet werden, die die notwendigen Vergleichs- und Hashoperationen definiert. Bei dieser *Orig-Generalisierung* kann die Objektverwaltung viele Operationen auf dieser generischen Struktur ausführen und nur diejenigen Operationen zu den Implementierungsklassen delegieren, die auf den konkreten Orig-Typ angewiesen sind. Im vorliegenden Prototyp wurde jedoch ein anderer Ansatz verfolgt: Da Konvertierungsobjekte außer ihrer Schlüsselinformation (also den Rückverweisen) keine weiteren Daten enthalten, können sie immer sofort instanziiert werden, da eine Bedarfsaktivierung keinen Vorteil mehr bietet. In dieser Hinsicht entspricht die Konvertierungs-Objektverwaltung der Hauptspeicher-Objektverwaltung, da Aktivierung und Deaktivierung entfallen. Nachteilig ist jedoch, daß bei der Anfrageauswertung der Extent durchsucht werden muß, um festzustellen, ob für einen bestimmten Schlüsselwert bereits ein Objekt existiert. Bei einer Neuimplementierung sollte daher der Orig-Generalisierung der Vorzug gegeben werden.

22. Ein ähnliches Prinzip liegt den ONTOS Storage-Managern zugrunde. Während der Projektlaufzeit standen diese aber noch nicht zur Verfügung, so daß dieses eigene Verfahren von mir entwickelt werden mußte.

d) Der realisierte Prototyp ist ein reines Hauptspeichersystem, bei dem alle persistenten Objekte von externen Datenbanken importiert werden. Wird das Sichtsystem direkt in einer Datenbank implementiert, können die eigenen persistenten Objekte durch eine zusätzliche *Datenbank-Objektverwaltung* eingebunden werden. Als Schlüsselwert dient in einem solchen Fall ein von der jeweiligen Datenbank vergebener Objektidentifier.

Im ODMG-Modell ist für jede Klasse ein Extent definiert, der die Menge aller Instanzen dieser Klasse umfaßt. Durch die Möglichkeit, diesen Extent nur teilweise im Speicher zu materialisieren, muß man neben diesem *vollen Extent* noch zwei weitere Teilmengen hiervon in Betracht ziehen: den aktiven und den sichtbaren Extent.

Definition 4.9: voller, aktiver und sichtbarer Extent

Der *(volle) Extent* umfaßt alle zu einer Klasse gehörenden Objekte, unabhängig davon, ob sie bereits instanziiert sind oder nicht. Der *aktive Extent* umfaßt alle Objekte des Extents, die sich bereits im Speicher befinden, also aktiviert worden sind. Der *sichtbare Extent* umfaßt all jene Objekte des vollen Extents, für die bereits eine Referenz angelegt worden ist, die also entweder schon aktiviert sind, oder durch Dereferenzierung jederzeit aktiviert werden können. Es gilt:

$$\text{aktiver Extent} \subseteq \text{sichtbarer Extent} \subseteq \text{voller Extent}$$

Im implementierten System stehen Zugriffsfunktionen für den vollen und den aktiven Extent zur Verfügung. Sie werden im Zusammenhang mit dem Data Dictionary (Anhang H.4) und den Basisklassen (Anhang H.7) erläutert.

Beispiel 4.6:

Bild 12 zeigt ein Beispiel für den Objektzugriff durch eine Referenz. Das Referenzobjekt (Ref_Any) enthält den Verwaltungsindex 2 und die Referenznummer 15. Der Liste aller Verwaltungsobjekte ist zu entnehmen, daß die Importverwaltung (ImpRefHdlr) für diese Referenz zuständig ist. Diese führt eine Tabelle, in der für die Referenznummer 15 die zur Aktivierung des Objekts notwendige Schlüsselinformation (keyval = 'key c') eingetragen ist. Hier ist das Objekt aber bereits aktiviert und der Zeiger auf das Objekt kann direkt der Tabelle entnommen werden. Ist der Zeiger noch nil, wie etwa bei Referenznummer 1, so wird gegebenenfalls die Aktivierung angestoßen und der Zeiger auf das neue Objekt in die Tabelle eingetragen. ■

Bei der Interaktion von Referenzen, Verwaltungsobjekten und der Datenbank lassen sich die folgenden Arbeitsschritte unterscheiden: Referenzerzeugung, Dereferenzierung, Aktivierung, Deaktivierung, Erzeugung und Löschung. Diese werden im folgenden kurz beschrieben. Für den Zugriff auf den Extent sowie die Konvertierung und Zuordnung von Indexwerten, Schlüsselwerten und Objekten sind zusätzliche Basisfunktionen definiert, die in Anhang H.2 zu finden sind.

Referenzerzeugung

Bei der Referenzerzeugung wird für ein bestimmtes Objekt eine neue Referenz angelegt. Hierbei gibt es unterschiedliche Arten, das Zielobjekt zu identifizieren:

a) durch eine andere Referenz auf das Objekt: Die XRID der anderen Referenz kann direkt kopiert werden. Ein Zugriff auf das Objekt selbst ist nicht notwendig.

b) durch das Objekt selbst: Für diesen Fall wird in jedem Objekt eine Referenz auf sich selbst abgelegt, die wie in Fall a) kopiert werden kann (siehe Anhang H.7, Pobject::_selfRef).

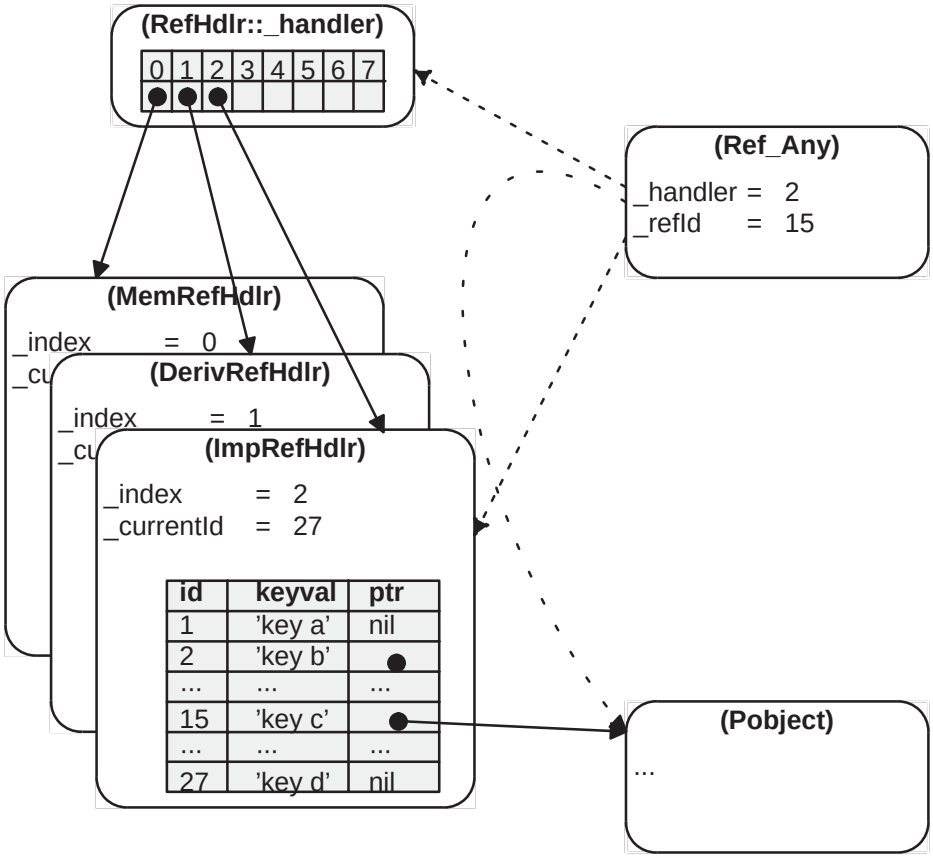


Bild 12: Referenzauswertung mit mehrfachen Verwaltungsobjekten

c) durch einen Schlüsselwert: Eine solche Referenzerzeugung muß stets an ein konkretes Verwaltungsobjekt gerichtet werden. Ist der gesuchte Schlüsselwert bereits in der Tabelle enthalten, wird aus dem Verwaltungsindex und der zugehörigen RID eine Referenz erzeugt. Andernfalls wird eine neue Referenznummer erzeugt und mit dem Schlüsselwert in die Tabelle eingetragen. Ein Sonderfall ist die Erzeugung einer Referenz für ein neu zu erzeugendes nicht-virtuelles Hauptspeicherobjekt, bei der ohne einen Schlüsselwert eine neue Referenznummer vergeben wird.

Der entsprechende Algorithmus create_reference ist in Anhang H.2 angegeben.

Dereferenzierung

Bei der Dereferenzierung wird das zu einer erweiterten Referenznummer gehörige Objekt bestimmt. Die Referenz bestimmt mit Hilfe der VID das entsprechende Verwaltungsobjekt und übergibt ihm die Referenznummer. Ist das Objekt bereits aktiviert, kann es direkt zurückgeliefert werden. Anderenfalls muß das Verwaltungsobjekt zuvor das Zielobjekt mit Hilfe der Schlüsselinformation aktivieren und in die interne Tabelle eintragen. Der Algorithmus get_object ist ebenfalls im Anhang zu finden.

Aktivierung

Wenn über eine Referenz auf ein Zielobjekt zugegriffen werden soll, das sich noch nicht im Speicher befindet, so wird bei der Dereferenzierung automatisch die Aktivierung angestoßen. Bei der Aktivierung wird die übergebene Schlüsselinformation verwendet, um im Speicher das zugehö-

rige Objekt zu erzeugen und mit Daten zu füllen. Der konkrete Ablauf einer solchen Aktivierung hängt vom jeweiligen Verwaltungsobjekt ab.

a) Die *Hauptspeicher-Objektverwaltung* unterstützt keine Objektaktivierung. Da sie nur Objekte verwaltet, die sich bereits im Speicher befinden, führt jede vom MemRefHdlr verwaltete Referenz immer auf ein existierendes Objekt, so daß die Aktivierungsfunktion nie aufgerufen werden kann.

b) Bei der *Import-Objektverwaltung* besteht der Schlüsselwert immer aus einer GOID. Mit dieser Zeichenkette wird systemweit eindeutig das externe Objekt bestimmt, von dem das Importobjekt abgeleitet worden ist. Zwei GOIDs verweisen dann und nur dann auf dasselbe externe Objekt, wenn die Zeichenketten identisch sind. Eine GOID besteht aus einem Namen zur Identifikation der externen Datenbank, dem Klassennamen des externen Objekts und der lokalen OID (LOID) des Objekts. Sie hat das folgende Format:

<GOID> == <cnt1><clsname><cnt2><loid><ldaid>
<cnt1> == Länge von <clsname> als dreistellige Dezimalzahl
<clsname> == externer Klassenname des Objekts
<cnt2> == Länge von <loid> als dreistellige Dezimalzahl
<loid> == lokale, proprietäre Objektkennung der externen Datenbank
<ldaid> == Identifikationskennung der externen Datenbank

Da C++ keine Objektmigration unterstützt, muß das Importobjekt direkt in der spezifischsten Importklasse erzeugt werden, zu der es gehört. Weil statische Typinformationen aus der Referenz hierfür nicht ausreichend sind, enthält jede GOID den Namen der spezifischsten exportierten Klasse des externen Objekts, aus dem mit Hilfe des Data Dictionarys die zugehörige Importklasse bestimmt wird. Über eine Netzwerkkomponente wird dann mit der Datenbankkennung eine Verbindung zur externen Datenbank aufgebaut und die lokale Oid übertragen. Dann werden die zu importierenden Datenfelder des externen Objekts empfangen und an eine geeignete Konstruktorfunktion der Importklasse übergeben. Enthält das Objekt Verweise auf andere Objekte, werden die hierbei übertragenen GOIDs in Referenzen umgewandelt (*Internisierung*). Im IRO-DB-System besteht die Datenbankkennung aus dem Hostnamen, dem Datenbanknamen und einem Schlüsselwort für den Datenbanktyp (O_2, Ontos, Ingres, ...). Diese Angaben adressieren einen sogenannten *Local Database Adapter (LDA)*, der die externe Datenbank an den *Communication Layer*, eine objektorientierte Erweiterung von SQL/CLI, anschließt. In anderen Systemen kann man statt dessen JDBC/ODBC-Lokatoren oder globale Namen verwenden. Die LOID wird von der externen Datenbank verwendet, um das Objekt zu identifizieren. Ihr Inhalt ist vom externen Datenbanksystem frei wählbar, die LOID muß für ein Objekt aber eindeutig sein und als Zeichenkette darstellbar sein. Der vollständige Algorithmus zur Objektaktivierung ist im Anhang zu finden.

c) Die *Konvertierungs-Objektverwaltung* stellt nur im Falle der Orig-Generalisierung Aktivierungsfunktionalität zur Verfügung. Genau wie bei der Importobjektaktivierung muß aus dem Schlüsselwert die spezifischste Klasse bestimmbar sein, in der das Objekt erzeugt werden muß. Im Gegensatz zu Importobjekten kann dann aber die Schlüsselinformation direkt an die Konstruktorfunktion übergeben werden.

d) Bei der *Datenbank-Objektverwaltung* ist der verwendete Schlüssel immer ein von der Datenbank vergebener Identifikator. Mit diesem kann direkt eine von der Datenbank zur Verfügung gestellte Funktion zur Objektaktivierung aufgerufen werden. Ontos stellt zum Beispiel den Datentyp OC_Reference zur Verfügung, über dessen getReferent-Methode ein

Objekt aktiviert werden kann. Zusätzlich kann man benannte Objekte auch über OC_Lookup aktivieren ([ONT94b]).

Deaktivierung

Entsprechend der Aktivierung wird auch die Deaktivierung nur für Import- oder Datenbankobjekte aufgerufen. Als erster Schritt müssen etwaige Änderungen am Objekt in die (eigene bzw. externe) Datenbank zurückgeschrieben werden. Dann wird in den internen Tabellen der Adreßeintrag für das Objekt gelöscht, und das Objekt kann mit Hilfe des Destruktors aus dem Speicher entfernt werden. Die vergebene Referenznummer bleibt weiterhin gültig; bei erneutem Zugriff wird das Objekt automatisch wieder aktiviert.

Erzeugung

Bei der Objekterzeugung handelt es sich um einen Datenbankupdate. OQL erlaubt durch einen Konstuktoraufruf, neue Objekte zu erzeugen und mit Attributwerten zu belegen. Um dies bei virtuellen Objekten durchzuführen, wird der Ablauf in zwei Schritte zerlegt. Zuerst werden die nötigen Operationen ausgeführt, um das Objekt in der Datenbank zu erzeugen; dann wird das Objekt aktiviert. Um diese Trennung zu ermöglichen, wird zur Objekterzeugung nicht direkt ein Konstruktor aufgerufen, sondern eine in der Klasse definierte Objekterzeugungsmethode. Für die verschiedenen Objektarten gibt es die folgenden Vorgehensweisen:

a) Für *Hauptspeicherobjekte* ist die Zweiteilung nicht notwendig. Die Erzeugungsfunktion kann direkt einen Konstruktor aufrufen und die übergebenen Attributwerte zuweisen. Zuvor können gegebenenfalls die Vollständigkeit der Attributliste und etwaige Schlüsselbedingungen überprüft werden.

b) Bei *Importobjekten* muß das entsprechende Objekt in der externen Datenbank erzeugt werden. Hierzu wird aus der übergebenen Attributliste eine OQL-Anfrage erzeugt und an die externe Datenbank übertragen. Der in der Anfrage enthaltene Konstruktoraufruf liefert ein externes Objekt, also eine GOID zurück, das in eine Referenz umgewandelt und dann zurückgeliefert wird. Eine Aktivierung des Importobjekts ist zu diesem Zeitpunkt nicht notwendig. Die Erzeugungsmethode kann generisch in ImportClass definiert oder für jede Klasse einzeln zur Verfügung gestellt werden. Im Prototyp handelt es sich um eine einzelne generische Funktion, die zusätzlich für jede Importklasse die erzeugten OQL-Anfragen zur Wiederbenutzung abspeichert.

c) Bei *Konvertierungsobjekten* kann kein allgemeines Verfahren zur Objekterzeugung angegeben werden. Es hängt jeweils von der konkreten Klasse ab, wie durch geeignete rekursive Erzeugung von Basisobjekten ein neues Konvertierungsobjekt erzeugt werden kann. Die entsprechende Implementierung muß im Rahmen der Sprachbindung zur Verfügung gestellt werden und wird dann von der Objektverwaltung aufgerufen.

d) Bei *Datenbankobjekten* wird das von der Datenbank angebotene Verfahren zur Objekterzeugung verwendet. Je nach Datenbank kann es sich hierbei um eine OQL-Anfrage, einen Funktionsaufruf oder einen Konstruktoraufruf handeln.

Löschung

Auch bei der Objektlöschung müssen Datenbankoperationen abgesetzt werden, um das Objekt dauerhaft zu entfernen. Bei *Hauptspeicherobjekten* muß nur das Objekt selbst gelöscht werden, da es in keiner Datenbank vorliegt. Bei *Datenbankobjekten* wird eine von der Datenbank ange-

botene Löschfunktion aufgerufen. *Importobjekte* müssen deaktiviert und aus der externen Datenbank entfernt werden. Da OQL keine Löschoperationen bietet, muß diese Funktionalität durch die Datenbankschnittstelle (LDA) zur Verfügung gestellt werden. Bei *Konvertierungsobjekten* muß die Löschoperation genau wie die Erzeugungsoperation explizit implementiert werden. Im allgemeinen wird sie das Konvertierungsobjekt deaktivieren und rekursiv die Löschfunktionen der Basisobjekte aufrufen. Abhängig von der Klassendefinition können aber auch andere Operationen notwendig sein, wie zum Beispiel das Löschen einzelner statt aller Basisobjekte oder das Verändern oder Löschen von Attributwerten. In allen Fällen muß nach dem Löschen eines Objekts der entsprechende Eintrag in den Verwaltungsobjekten als ungültig markiert werden, damit verbleibende Referenzen nicht versuchen, auf das gelöschte Objekt zuzugreifen.

4.4 Zusicherung referenzieller Integrität bei Relationships

Der vorgestellte Referenzmechanismus erlaubt es, Zeiger auf Objekte zu verwalten. Für Relationships zwischen Klassen reicht dies jedoch nicht aus. Gemäß ODMG–93 sind Relationships stets bidirektional; jedes der beteiligten Objekte erhält also eine Referenz auf das jeweils andere Objekt. Jede Veränderung einer solchen Zuordnung muß folglich stets beide Objekte verändern. Wenn also der Person in Bild 13a eine Adresse zugewiesen wird, muß zusätzlich diese Person im resident-Feld der Adresse abgelegt werden. Diese indirekten Zugriffe können sich sogar noch auf weitere Objekte ausdehnen, wie Bild 13b zeigt: Um dem Objekt P_1 die Adresse A_2 zuweisen zu können, müssen zuvor die bereits bestehenden Verbindungen zwischen P_1 und A_1 bzw. P_2 und A_2 aufgelöst werden.

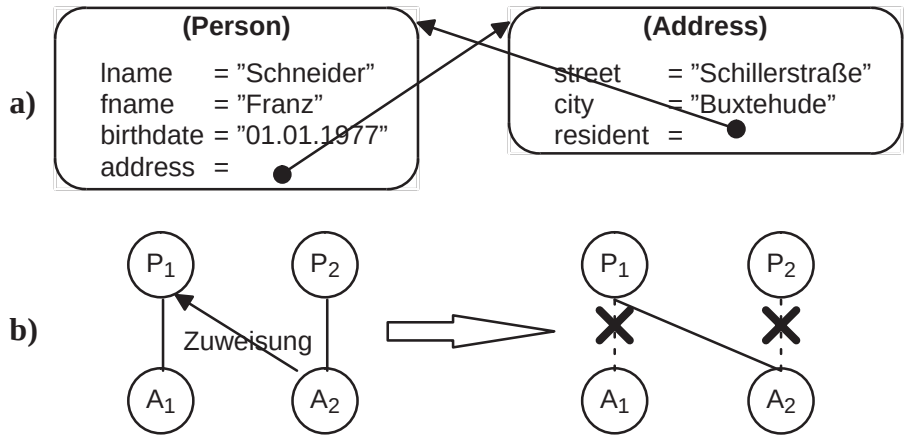


Bild 13: Inverse Pfade bei Relationships

Aus diesem Grund werden die Relationship-Pfade nicht als einfache Referenzen in den Objekten abgelegt, sondern als spezielle eingebettete Pfadobjekte. Dies entspricht auch der Sprachbindung, die entsprechende Objekte mit eigenen Methoden definiert, wobei jedoch die konkrete Implementierung, vor allem das Verfahren zur Sicherstellung referenzieller Integrität, dem Implementierer überlassen bleibt.[23] In jedem Pfadobjekt müssen dabei die folgenden Informationen abgelegt werden:

a) Die Zielobjekte: Jedes Pfadobjekt enthält eine Referenz auf das Zielobjekt, bzw. eine Menge oder Liste solcher Referenzen. Diese beschreiben das Ziel der Relationship.

b) Informationen zu den inversen Pfadobjekten: Um das inverse Pfadobjekt anpassen zu können, reicht es nicht aus, das Zielobjekt selbst zu kennen. Man muß auch wissen, wo sich in

23. Version 2.0 des Standards ist hier detaillierter; siehe Abschnitt 6.2.2.

diesem Zielobjekt die Inverse befindet, um dieser den neuen Wert zuweisen zu können. Das Ablegen eines direkten Verweises auf das inverse Pfadobjekt hat jedoch einige entscheidende Nachteile: i) das referenzierte Objekt muß aktiviert sein, damit das Pfadobjekt überhaupt existiert, ii) da beide Pfadobjekte das jeweils andere als Inverse haben, müssen die Konstruktoren geeignet koordiniert werden, iii) bei mehrwertigen Relationships müssen Verweismengen und -listen verwaltet werden, iv) bei jedem Update müssen neben den eigentlichen Zielen auch die Verweise geändert werden. Um diese Nachteile zu umgehen, wird im Pfadobjekt statt der direkten Verweise eine Funktion abgelegt, die für ein Zielobjekt das zugehörige Pfadobjekt bestimmt. Diese Funktion wird als generischer Funktionszeiger übergeben und kann daher von der Zielklasse geeignet implementiert werden.

c) Ein Zeiger auf das Trägerobjekt: Das Datenbankobjekt, in das das Pfadobjekt eingebettet ist, wird als dessen *Trägerobjekt* bezeichnet. Der Träger wird als Zielobjekt für die Inverse benötigt und ermöglicht zusätzlich die automatische Ausführung von mark_modified.

Neben Operationen zur Traversierung des Pfads werden Operationen zum Setzen, Hinzufügen und Entfernen von Zielobjekten angeboten. Bei der Implementierung solcher Pfadobjekte muß man eine Reihe von Kriterien in Betracht ziehen. Im folgenden gehe ich auf die einzelnen Kriterien ein und beschreibe den verwendeten Lösungsansatz.

Einfluß der eigenen Wertigkeit

Jeder Pfad ist entweder einwertig, mengenwertig oder listenwertig. Jede dieser Wertigkeiten benötigt eine eigene Implementierung, da sich ihre Funktionalität und somit die Zugriffsmethoden voneinander unterscheiden. Die bereitzustellenden Funktionen werden durch den Standard festgelegt ([Cat94], Kapitel 5.3.2.2) und definieren die Klassen SimpleRelAny, SetRelAny und ListRelAny.

Typsicherheit

Während die genannten Klassen intern mit ungetypten Referenzen arbeiten, müssen die bei der Traversierung gelieferten Objekte und die zum Update übergebenen Parameter vom Objekttyp der referenzierten Klasse sein. Dies geschieht wie bei einfachen Referenzen durch Templates (SimpleRel<T>, SetRel<T>, ListRel<T>), die von den typunabhängigen Implementierungen abgeleitet werden.

Einfluß der inversen Wertigkeit

Zur Erhaltung referenzieller Integrität müssen auf dem inversen Pfadobjekt automatisch Einfüge- und Löschoperationen ausgeführt werden. Prinzipiell hängen diese Operationen von der Wertigkeit des inversen Pfads ab, da sich die notwendigen Aktionen für einwertige, mengenwertige und listenwertige Pfade unterscheiden. Eine Untersuchung dieser Aktionen zeigt, daß alle Änderungen am inversen Pfad durch zwei Elementaroperationen simpleRemove und simpleInsert darstellbar sind. Die Implementierung dieser Funktionen hängt zwar noch von der lokalen Wertigkeit ab, der Aufruf kann aber einheitlich über virtuelle Funktionen der generischen Superklasse RelPathImpl erfolgen. Durch diesen Ansatz hat die inverse Wertigkeit keinen Einfluß mehr auf die Pfadimplementierung. Die genannten Basisfunktionen und einige Beispiele für ihren Einsatz sind in Anhang H.3 zu finden.

Einfluß der eigenen Kategorie (Sichtspezifische Erweiterung)

Das bisher gesagte gilt hauptsächlich für normale Klassen, bei denen die Relationships direkt als Referenzen bzw. Referenzmengen oder -listen am Objekt abgelegt sind. Diese Implementierung

kann größtenteils unverändert für Importklassen übernommen werden, da sie sich durch die Zwischenspeicherung aller Properties wie normale Klassen verhalten. Updates werden nicht direkt an die zugrunde liegende Datenbank weitergeleitet, sondern am Cache vorgenommen und erst zu bestimmten Synchronisationszeitpunkten in die Datenbank zurückgeschrieben. Im Unterschied zu normalen Klassen benötigen Import-Relationships aber eine Initialisierungsfunktion, mit der die Anfangsreferenzen zugewiesen werden können, ohne einen Update oder eine Inversenkorrektur auszulösen. Ebenfalls müssen bei der Deaktivierung alle Referenzen gelöscht werden, ohne die Inverse zu verändern. Bei Konvertierungsklassen muß jede Relationship eine eigene Implementierung erhalten. Beim Lesen muß die in der Spezifikation definierte Relationshipklausel ausgewertet werden; beim Schreiben müssen, sofern dies überhaupt möglich ist, entsprechende Manipulationen an den Basisobjekten vorgenommen werden, damit die Änderung wie gewünscht in der abgeleiteten Relationship sichtbar wird. Auf Grund der Komplexität solcher Ausdrücke müssen Updates von abgeleiteten Relationships grundsätzlich ausprogrammiert werden. Um den Berechnungsaufwand bei der Traversierung zu verringern, kann das Zielobjekt, bzw. die Zielmenge oder -liste, nach der ersten Berechnung in einer lokalen Variablen zwischengespeichert werden. Vor jedem weiteren Zugriff muß dann allerdings die Korrektheit des aktuell gespeicherten Wertes überprüft werden.

Um die eigentliche Implementierung vor dem Anwender zu verbergen, dürfen die kategorieabhängigen Klassen keinen Einfluß auf das in der Sprachbindung sichtbare Schema haben. Aus diesem Grunde wird der Zugriff auf Relationships von deren Implementierung getrennt. Für jede Wertigkeit existiert genau eine *Zugriffsklasse*, die die vom Standard geforderten Funktionen zur Verfügung stellt. Diese Klassen stellen den sichtbaren Teil der Relationships dar, und ihre Instanzen werden in die Datenbankobjekte eingebettet. Die eigentliche Funktionalität hinter diesen Zugriffen wird dann durch zusätzliche *Implementierungsklassen* zur Verfügung gestellt. Jedem Zugriffsobjekt wird zum Zeitpunkt seiner Erzeugung ein entsprechendes Implementierungsobjekt zugeordnet, das alle notwendigen Informationen über die Relationship enthält. Alle Zugriffe auf die Relationship werden vom Zugriffsobjekt an das Implementierungsobjekt weitergeleitet.

Beispiel 4.7:

Bild 14 zeigt die notwendigen Instanzen bei einer 1-zu-1-Relationship.[24] Die Person und die Adresse enthalten jeweils ein eingebettetes Zugriffsobjekt. Dieses stellt alle Operationen zur Verfügung, die für einwertige Pfade definiert sind, und ist unabhängig von der jeweiligen Kategorie der beteiligten Klassen. Die eigentliche Funktionalität ist in den zugehörigen Implementierungsobjekten vom Typ SimpleRelImpl realisiert. Diese Klasse ist spezifisch für einwertige Relationships zwischen normalen Klassen und verwaltet die notwendigen Zeiger auf das Ziel, den eigenen Träger und das inverse Implementierungsobjekt. ■

Einfluß der inversen Kategorie (Sichtspezifische Erweiterung)

Durch die Verwendung normierter Elementaroperationen und durch die Trennung von Zugriff und Implementierung hat die Kategorie der inversen Klasse weder einen Einfluß auf die Zugriffsklassen noch auf die Implementierungsklassen.

Zusammenfassung

Durch die beschriebene Parametrisierung, die Trennung von Schnittstelle und Implementierung sowie die Definition von Elementaroperationen ist es nun möglich, fast alle Pfadimplementie-

24. Genestete Objekte und Methodendarstellung in Objekten sind in [RBP*91] nicht vorgesehen, sind hier aber anschaulicher als die dort verwendeten Aggregations- und Instanziierungskanten.

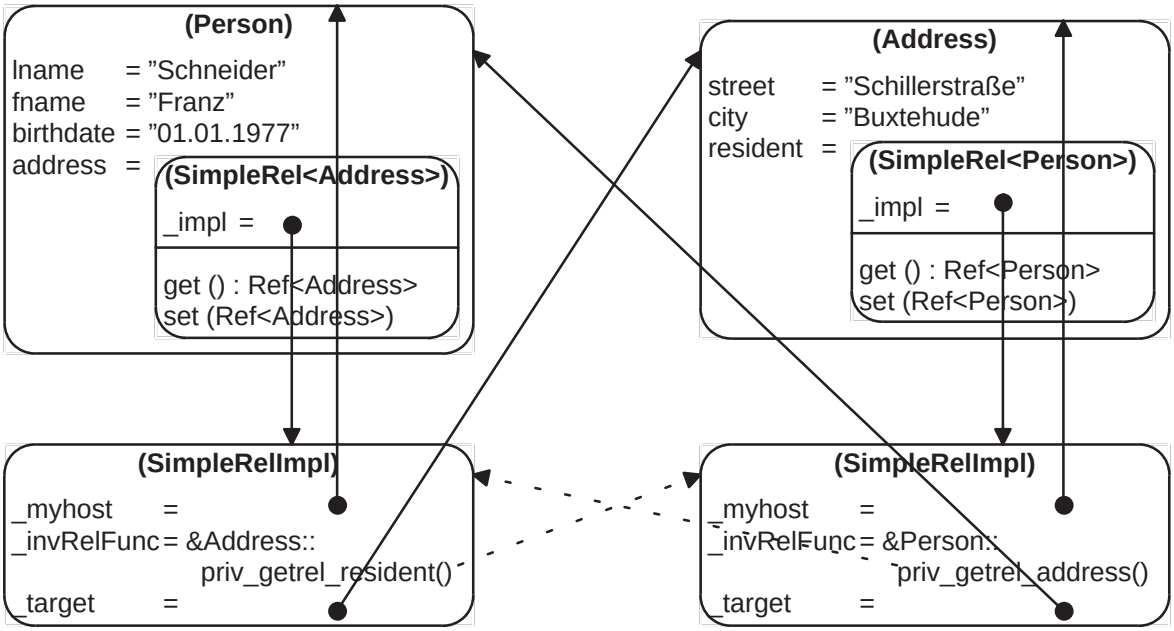


Bild 14: Implementierungsklassen für Relationships

rungen vorzudefinieren oder automatisch mit Templates zu generieren. Nur die schemaspezifischen abgeleiteten Pfade benötigen explizite Implementierungen. In Anhang H.3 ist die vollständige Klassenhierarchie wiedergegeben.

4.5 Generischer Datenzugriff zur Anfrageunterstützung

Die bisher vorgestellte Sprachbindung und die Objektverwaltung stellen Anwendungsprogrammen die Sichten in Form von C++-Objekten zur Verfügung, die mit den üblichen in C++ vorhandenen Sprachelementen manipuliert werden können. Neben kompilierten Anwendungen stellt aber eine deklarative Anfragekomponente ein weiteres wesentliches Element eines Datenbanksystems dar. Die Anfragebearbeitung kann man in drei Phasen einteilen: Beim *Parsen* wird der Anfragetext analysiert und in einen Operatorbaum überführt. Beim *Umformen* wird der Baum optimiert, und Zugriffe auf Sichtobjekte werden, soweit möglich und erforderlich, entsprechend der Sichtdefinition durch Zugriffe auf die zugehörigen Basisobjekte ersetzt (Anfrageexpansion). Weiterhin kann im Falle einer verteilten Datenbank die Anfrage in Teilanfragen für die einzelnen Datenbanken zerlegt werden (siehe [FSF95]). Nach diesen vorbereitenden Schritten erfolgt bei der *Auswertung* der eigentliche Datenzugriff für die Ergebnisberechnung. Für diese Aufgaben werden zusätzliche Teilkomponenten benötigt, die in den folgenden Abschnitten vorgestellt werden: Ein *Data Dictionary*, in dem sicht- und klassenspezifische Informationen abgelegt werden, ein *generisches Datenformat* zur Darstellung von Zwischenergebnissen sowie eine Reihe von *Wrapperfunktionen* zum generischen Zugriff auf Objekte der Sprachbindung.

4.5.1 Das Data Dictionary

Um Anfragen überprüfen und auswerten zu können, benötigt das Anfragesystem Informationen über das Datenbankschema. Diese Metadaten sind üblicherweise im sogenannten *Data Dictionary* persistent in der Datenbank abgelegt. Da ODMG–93 kein eigenes Data Dictionary definiert, mußte für das Sichtsystem eines entworfen werden. Neben den reinen ODMG-bezogenen Daten müssen dabei zusätzlich die folgenden sichtspezifischen Informationen abgelegt werden:

- ein Verweis auf das *orig-Attribut* von virtuellen Klassen

- eine Markierung *is_private*, um Attribute als privat zu kennzeichnen
- die *Kategorie* der Klasse
- die *Herleitungsklauseln* für Extents, Properties und Methoden in OQL-Format
- der Name und die Startadresse von *Wrapperfunktionen* für Methoden und Konstruktoren
- diverse *Hilfsstrukturen* und *Hashtabellen* zur Zugriffsbeschleunigung

Eine detaillierte Beschreibung des Data Dictionary erfolgt in Anhang H.4; eine vollständige Spezifikation mit allen Schnittstellen, Klassendefinitionen und Diskussionen ist im IRO–DB-Dokument [D4-3/1] gegeben.

Dynamische Erzeugung der Metadaten

Da das hier vorgestellte Sichtsystem vorrangig für Integrationsumgebungen ausgelegt ist, bin ich beim Entwurf davon ausgegangen, daß auf der integrierten Seite keine eigene Datenbank zur Verfügung steht. Das Data Dictionary mit dem Sichtschema muß daher dynamisch im Hauptspeicher aufgebaut werden. Anstatt die Schemadefinitionen zur Laufzeit erneut einzulesen, wird im Rahmen der Sprachbindung eine zusätzliche Funktion erzeugt, die das Data Dictionary ohne erneutes Parsen füllt. Dies hat die folgenden Vorteile:

- die ODL-Dateien werden zur Laufzeit nicht mehr benötigt
- das Metaschema ist immer konsistent mit dem generierten Code der Sprachbindung
- der Parser muß nicht im Laufzeitsystem enthalten sein (verminderte Codegröße, kein Zeitaufwand für das Parsen)
- die Startadressen der Wrapperfunktionen können direkt in das Data Dictionary eingetragen werden, so daß die dynamische Symbolsuche entfällt
- die statische Typinformation der Sichtklassen kann initialisiert werden: Jede Klasse bekommt eine statische Variable _metaclass, die auf die zugehörige Klassendefinition im Data Dictionary zeigt. Hierdurch erhalten alle Objekte Laufzeitinformationen über ihren Typ.

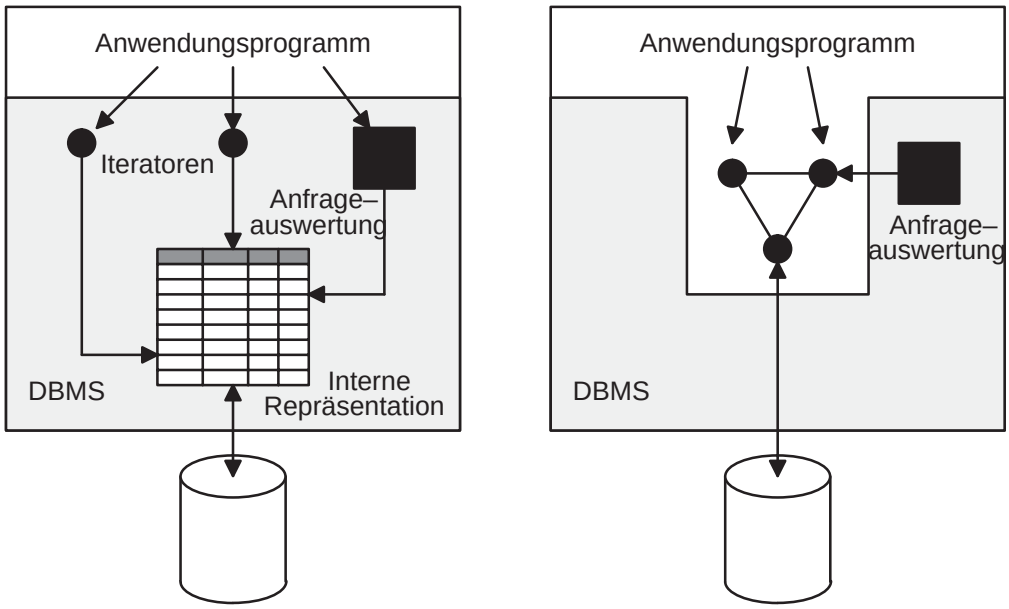
Der schematische Aufbau einer solchen Funktion ist ebenfalls in Anhang H.4 angegeben; sie wird automatisch durch den Präprozessor generiert.

4.5.2 Generische Datenrepräsentation

Während der Anfrageauswertung muß viel mit Zwischenergebnissen gerechnet werden, deren Strukturen zur Kompilationszeit noch nicht bekannt sind und auf die generisch zugegriffen werden muß. Da diese Strukturen von außen nicht sichtbar sind, kann man hierfür die Vorteile einer indirekten Abbildung des Datenmodells ausnutzen und muß nicht auf Binärkompatibilität mit den Programmdaten achten. In IRO-DB ist für den schemaunabhängigen Datentransport in der Kommunikationsschicht bereits eine solche generische Datenrepräsentation entwickelt worden ([Leg96]). Dieses *Generische Datenformat* (kurz *GF*) kann direkt für die Anfrageauswertung weiterverwendet werden. Es besteht aus einer vordefinierten Menge von Klassen, mit denen jeder Datentyp modelliert werden kann. Die vollständige Abbildung der ODMG-Datentypen in dieses Datenformat ist in Anhang H.5 angegeben. Die generischen Datenstrukturen sind von Régis Legoff für die Kommunikationsschicht entworfen und implementiert worden. Einige Rohentwürfe, sowie die Anwendung der Strukturen für die Anfrageauswertung und die Wrapperfunktionen stammen von mir.

4.5.3 Hilfsfunktionen und Wrapper

Das generische Datenformat erlaubt zwar umfangreiche Manipulationen auf Daten, der Zugriff auf Objekte stellt jedoch weiterhin ein Problem dar. Bei relationalen Datenbanken liegen alle Daten optimiert in internen Strukturen der Datenbank vor; das Anfragesystem kann beliebig darauf arbeiten, und nur die reinen Ergebnisse werden an das Anwendungsprogramm zurückgegeben (Bild 15a). Das objektorientierte Modell hebt diese Datentrennung jedoch auf, so daß das Anfragesystem nicht mehr ausschließlich auf internen Strukturen operieren kann, sondern direkt auf Objekte zugreifen muß, die sich innerhalb des Anwendungsprogramms befinden (Bild 15b).



a) relational b) objektorientiert

Bild 15: Unterschied zwischen relationalem und objektorientiertem Zugriff

Im Gegensatz zum generischen Datenformat, das auf der indirekten Abbildung von ODL nach C++ basiert und somit generischen Zugriff auf alle Datenelemente zur Verfügung stellt, basiert die ODMG-Sprachbindung auf der direkten Abbildung. Nach der Kompilation bietet C++ keine Möglichkeit mehr, die vom Anfragesystem geforderten deklarativen, namensgebundenen Zugriffe auf die entsprechenden Speicherblockoperationen abzubilden.[25] Um dennoch dynamische Zugriffe zur Verfügung zu stellen, gibt es drei unterschiedliche Ansätze:

a) Separate Übersetzung:
Aus der deklarativen Anfrage wird ein C++-Quelltext erzeugt, der von einem Compiler übersetzt, dann gebunden und ausgeführt wird. Da der Compiler alle Speicherstrukturen kennt, stellen die Objektzugriffe nun kein Problem dar. Der offensichtliche Nachteil ist aber, daß jede Anfrage einen vollständigen Übersetzungslauf zur Ausführung benötigt.

25. Dieses Fehlen von Metainformationen und Zugriffsmethoden ist ein möglicher Grund für die geringe Funktionalität derzeitiger Anfrageimplementierungen für C++-basierte objektorientierte Datenbanken. ObjectStore kann zum Beispiel nur bereits existierende Objekte einer einzelnen Klasse mit Hilfe einer Bedingung selektieren ([ODI95]), Ontos kann zusätzlich statt der gefundenen Objekte eine Struktur mit ausgewählten Attributwerten zurückliefern ([ONT94a]). Eine etwas größere Teilmenge von OQL wird von POET zur Verfügung gestellt ([POE99]), allerdings fehlen bisher noch Umstrukturierungen und Methodenaufrufe. Das einzige vollständige Anfragesystem gehört derzeit zu O_2, den Entwicklern von OQL ([BDK92]), und steht nicht in C++ sondern nur in LISP zur Verfügung.

b) Direkte Adreßberechnung:
Kennt man das Abbildungsverfahren des Compilers sowie das vollständige Layout der Sichtklassen, so kann das Compilerverhalten nachempfunden werden und Feldadressen und -größen können zur Laufzeit für den Zugriff berechnet werden. Ein solches Verfahren wird zum Beispiel in der Ontos-Datenbank verwendet. Der Vorteil dieses Vorgehens ist, daß man sich den Kompilationsschritt bei der Anfrageauswertung erspart und trotzdem direkt auf den C++-Objekten arbeiten kann. Der Nachteil ist jedoch, daß man die Compilerdetails gut kennen muß, und man außerdem auf genau einen Compiler und eine Architektur beschränkt ist.

c) Vorkompilierte Zugriffsfunktionen:
Die dritte Alternative ist, die Sprachbindung um zusätzliche Zugriffsfunktionen zu erweitern, die eine generische Schnittstelle zu den spezifischen Daten darstellen. Da diese Funktionen kompiliert werden, wird die Abbildung auf die Speicherstrukturen automatisch vom Compiler übernommen, und da ihre Schnittstellen fest definiert sind, können sie ohne Probleme von der Anfragekomponente aus aufgerufen werden.

In meinem Sichtsystem bietet sich die letzte Alternative an, da die Sprachbindung fast vollständig von einem Präprozessor erzeugt wird, der auch die Erzeugung geeigneter Zugriffsfunktionen übernehmen kann (siehe Kapitel 4.6.1). Für das vollständige Zusammenspiel der Sprachbindung mit Anfrageauswertung und Objektverwaltung benötigt man Konstruktoren, Destruktoren und Zuweisungs-Operatoren für die Implementierungsklassen, freie Funktionen zur Datentypkonvertierung, virtuelle Zugriffsfunktionen, die über Superklassen aufgerufen werden, sowie sogenannte Wrapperfunktionen, die direkt über dynamische Funktionsaufrufe ausgeführt werden können. Im folgenden werden die einzelnen zu erfüllenden Aufgaben beschrieben; in Anhang H.6 findet man weitere Details.

Parameterübergabe an das Anfragesystem

Um Daten vom Anwendungsprogramm oder der Sprachbindung an das Anfragesystem übergeben zu können, muß für jeden im Schema vorkommenden Datentyp eine Konvertierungsroutine getGF erzeugt werden, die den kompilierten Wert von der Sprachbindung in das generische Datenformat umwandelt. Konvertierungsroutinen für die Basisdatentypen sowie Templates für Mengen- und Listentypen werden statisch als Bibliothek zur Verfügung gestellt.

Ergebnisrückgabe vom Anfragesystem

Die umgekehrte Situation ergibt sich bei der Rückgabe der Ergebnisse vom Anfragesystem an das Anwendungsprogramm sowie bei der Übergabe von Parametern bei Methodenaufrufen durch das Anfragesystem. Für jede getGF-Funktion wird also eine Inverse assignGF definiert, die ein gegebenes Datum aus dem generischen Format in die äquivalente C++-Struktur überführt.

Zugriff auf Objektfelder

Mittels getGF und assignGF können Datenstrukturen zwischen Anwendungsprogrammen und dem Anfragesystem ausgetauscht werden. Dieses Verfahren funktioniert jedoch nur für literale Datenwerte und nicht für Objekte. Objektreferenzen können zwar im GF-Format übergeben werden, bei der Dereferenzierung muß aber stets zum ursprünglichen Objekt aufgelöst werden, da Methodenaufrufe die C++-Speicherstruktur des Objekts erwarten. Um für die Anfrageauswertung generisch auf die Datenfelder eines Objekts zugreifen zu können, müssen zusätzliche Lese- und Schreibmethoden generiert werden. Zu jeder Klasse werden daher die Methoden

priv_get_property und priv_set_property hinzugefügt, die ein Attribut lesen oder schreiben, dessen Name als Parameter übergeben wird.

Methodenaufrufe

Da man in OQL Methoden innerhalb einer Anfrage aufrufen kann, braucht man auch geeignete Funktionen, um zur Laufzeit die korrekten Methoden zu finden und auszuführen. Hierfür muß eine Reihe weiterer Probleme gelöst werden:

a) Auswahl der korrekten Methode:
Der Methodenname und die Parameterliste reichen allein nicht aus, um die aufzurufende Methode eindeutig zu identifizieren. Da in ODMG alle Methoden virtuell sind, können sie in Subklassen neu definiert werden. Man muß also zusätzlich den Typ des ausführenden Objekts kennen und muß die zugehörige Klasse und alle Superklassen nach der spezifischsten Methode mit der gegebenen Signatur durchsuchen.

b) Bestimmung der Startadresse:
Wenn die aufzurufende Methode bekannt ist, muß die zugehörige Startadresse im Speicher gefunden werden, um die Methode auszuführen. C++ bietet hierfür zwar die Methode dlsym an, mit der dynamisch Symbole gebunden und lokalisiert werden können, man muß aber den internen Symbolnamen der Methode kennen, der bei der Übersetzung vom Compiler konstruiert wird. Da es mehrere solcher Name-Mangling-Verfahren gibt, führt die Nachbildung eines solchen Verfahrens wiederum zu einer Compilerabhängigkeit.

c) Parameterübergabe:
C++ bietet Funktionszeiger, mit deren Hilfe Funktionen zur Laufzeit dynamisch aufgerufen werden können. Diese Zeiger sind jedoch typgebunden; das heißt, daß zur Kompilationszeit bereits feststehen muß, welche Parametertypen an die Funktion übergeben werden sollen. Für jede Kombination von Parametertypen muß ein eigener Zeigertyp und ein eigener Aufruf zur Verfügung gestellt werden. Es können also nur Methoden aufgerufen werden, deren Signaturen zur Übersetzungszeit bekannt sind; der dynamische Aufruf beliebiger Funktionen zur Laufzeit ist mit diesem Verfahren nicht möglich. Hierzu müßte man auf Maschinenebene Stapelmanipulationen vornehmen, um die verschiedenen Parameter im C++-Format an die Methode zu übergeben. Dies erfordert umfangreiches Wissen über die Architektur und das Aufrufverhalten des vom Compiler generierten Codes, und das resultierende System ist genau an diese eine Konfiguration gebunden.

Auf Grund dieser Probleme ist es sehr schwer, innerhalb des Anfragesystems einen generischen Aufrufmechanismus für Methoden zur Verfügung zu stellen. Andererseits löst der Compiler all diese Aufgaben automatisch bei der Programmübersetzung. Durch den Präprozessor besteht somit auch hier die Möglichkeit, die für uns ‚schweren' Probleme dem Compiler zu überlassen und für die Interaktion mit dem Anfragesystem eine einfachere Schnittstelle zur Verfügung zu stellen. Hierzu wird für jede Methode im Schema eine sogenannte *Wrapper-Funktion* generiert. Sie übernimmt die Parameter im GF-Format, wandelt sie in flaches C++-Format, ruft die Funktion auf und wandelt das Ergebnis und eventuelle Ausgabeparameter wieder in das GF-Format zurück.

Objekterzeugung

Ein Spezialfall der Methodenaufrufe ist die Erzeugung neuer Objekte im Rahmen einer Anfrage. Hierzu muß eine statische Klassenmethode aufgerufen werden, die das Objekt in der Datenbank anlegt und durch einen Konstruktoraufruf als C++-Objekt zur Verfügung stellt (siehe Seite 90).

Diese create_object-Methode hat für alle Klassen dieselbe Signatur, da die initialen Attributwerte immer als GF-Struktur übergeben werden können. In der Methode werden gegebenenfalls Schlüsselbedingungen überprüft, dann wird ein Objekt mit den geforderten Attributwerten angelegt und zurückgegeben. Auch für diese Methode wird eine Wrapperfunktion erzeugt, die dynamisch vom Anfragesystem aufgerufen werden kann.

Die Implementierung von create_object unterscheidet sich für die verschiedenen Kategorien. Bei normalen Klassen kann einfach ein Konstruktor aufgerufen werden. Für Importklassen stellt der Objektmanager eine generische createObject-Methode zur Verfügung, die die zur Objekterzeugung notwendige OQL-Anfrage an die externe Datenbank schickt und die gelieferte Ergebnis-OID als Referenz zurückliefert. Für Konvertierungsklassen müssen in den zugrunde liegenden Basisklassen neue Instanzen angelegt werden, von denen das neue Konvertierungsobjekt abgeleitet wird. Da jede Objekterzeugung klassenspezifisch ist, muß die create_object-Methode vom Entwickler zur Verfügung gestellt werden. Die Wrapper werden automatisch generiert.

create_object sollte nur vom Anfragesystem verwendet werden, da das GF-Format für Anwendungsprogramme unbekannt ist. Will man dem Benutzer bzw. dem Anwendungsprogramm ebenfalls die Möglichkeit geben, neue Objekte zu erzeugen, so sollten hierfür zusätzliche Konstruktoren mit geeigneten Parameterlisten implementiert werden. Existiert ein solcher „Anwendungskonstruktor“ mit vollständiger Parameterliste, kann auch dieser vom Wrapper aus aufgerufen werden. Die Umwandlung vom GF-Format in C++-Format erfolgt dann im Wrapper.

Objektaktivierung

Neben der Erzeugung neuer Objekte muß auch das Aktivieren existierender Datenbankobjekte unterstützt werden. Hierzu wird ein zusätzlicher Konstruktor erzeugt, der die zur Aktivierung notwendige Parameterliste übergeben bekommt, das Objekt erzeugt und anschließend im aktiven Extent seiner Klasse, also der Liste der im Hauptspeicher befindlichen Instanzen, einträgt.

Man beachte, daß die Anfrageauswertung zwischen den Aktivierungs- und Erzeugungskonstruktoren unterscheiden muß. Kommt in einer herkömmlichen OQL-Anfrage ein Konstruktor-Aufruf vor, so handelt es sich stets um eine Objekterzeugung. Innerhalb des Sichtsystems sind jedoch auch die Extentklauseln mit Hilfe von Konstruktoren definiert. Diese sind stets als Objektaktivierung zu interpretieren, da sie nur einen Zugriffsmechanismus auf (real oder virtuell) in der Datenbank bereits existierende Objekte darstellen. Alle Konstruktoraufrufe, die bereits in einer Benutzeranfrage auftreten, müssen demnach als Erzeugungskonstruktoren markiert werden, während die durch Expansion der Extentklauseln eingefügten Konstruktoren als Aktivierungskonstruktoren zu markieren sind (siehe Expansionsbeispiel auf Seite 58).

Deaktivierung und Löschen

Zum Entfernen eines Objekts aus dem Speicher werden zwei weitere Methoden generiert. Der Destruktor entfernt das Objekt aus dem Extent und gibt alle belegten Ressourcen frei. Vor dem Aufruf des Destruktors muß die private Funktion priv_readout aufgerufen werden, die alle aktuellen Attributwerte in eine GF-Struktur überträgt, damit sie vom Objektmanager in die Datenbank zurückgeschrieben werden können (Schreib-Caching, siehe Abschnitt 4.2.5.2 und Kapitel 5).

Für das dauerhafte Löschen eines Objekts aus der Datenbank sind zuvor andere Operationen notwendig. Bei normalen Klassen ist der Destruktoraufruf gleichbedeutend mit einer Objektlöschung, da keine Aktivierungsmechanismen unterstützt werden. Bei Importobjekten bietet die Objektverwaltung einen generischen Löschvorgang, indem ein entsprechendes Kommando an

die externe Datenbank abgesetzt wird. Bei Konvertierungsklassen hingegen ist die Objektlöschung, genau wie die Objekterzeugung, für jede einzelne Klasse separat zu implementieren. Hierzu dient die Methode priv_destroy, die die notwendigen Operationen durchführt.

4.5.4 Zusammenspiel der Komponenten bei der Anfrageauswertung

Das folgende Beispiel zeigt, wie die gerade beschriebenen Zugriffsfunktionen verwendet werden, um eine Anfrage auszuführen.

Beispiel 4.8:

Gegeben sei die folgende Anfrage aus einem Anwendungsprogramm:

```
Set<char*>  res;
oql (res, "select a.lname from a in pubdb_authors where a.pubcount () > $1i", 10);
```

Die OQL-Funktion übergibt den Anfragetext und die Parameterliste an die eigentliche Anfrageauswertung. Der übergebene Parameter wird zuvor in das generische Datenformat umgewandelt, und das gelieferte Ergebnis wird anschließend aus dem generischen Format nach C++ zurückgewandelt. Der Ablauf der Funktion sieht prinzipiell also wie folgt aus:[26]

```
params = new GFList;                    // Parameter in internes Format wandeln
params→insert (getGF (10));
result = new GFSet;                     // leere Ergebnismenge in internem Format erstellen
evaluate_query ("select a.lname from a in pubdb_authors where a.pubcount () > $1i",
                result, params);        // Anfrageauswertung aufrufen
assignGF (res, result);                 // Ergebnismenge in C++-Format wandeln
```

Das Anfragesystem übersetzt den Anfragetext in einen Operatorbaum und führt die für jeden Knoten notwendigen Operationen aus. Hierbei werden die generischen Methoden mit den jeweils aktuellen Namen und Werten aufgerufen. Das folgende Bild zeigt schematisch den der Anfrage entsprechenden Operatorbaum und die während der Ausführung verwendeten Code-Fragmente. ■

26. Die angegebenen Codefragmente sollen nur den Ablauf illustrieren und entsprechen nicht dem real ausgeführten Code. Der Ausdruck „getGF(10)“, zum Beispiel, beschreibt nur, daß innerhalb der Anfrageauswertung erkannt wird, daß es sich bei dem Parameter um den Datentyp int handelt, der mittels GetGF in das generische Datenformat umgewandelt werden muß. Dies geschieht jedoch nicht durch spezifisch generierten Code, sondern im Rahmen umfangreicher Fallunterscheidungen bei der Parameterauswertung.

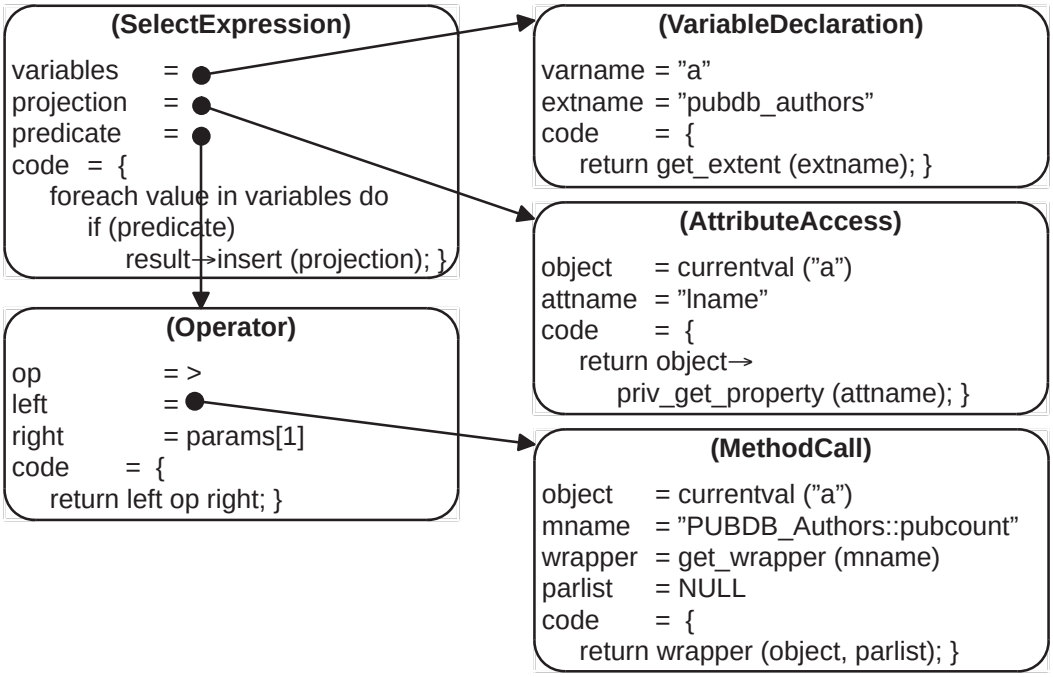


Bild 16: Operatorbaum mit Codefragmenten

4.6 Einbindung in das IRO–DB-System

Die bisher beschriebene Implementierung bezieht sich größtenteils auf die reine Sichtfunktionalität und kann prinzipiell in unterschiedlichen Umgebungen zum Einsatz kommen. An einigen Stellen sind jedoch bereits Abhängigkeiten mit anderen Komponenten sichtbar geworden. Im folgenden wird daher beschrieben, wie die beschriebenen Komponenten mit den anderen Modulen des IRO–DB-Prototypen zusammenarbeiten. Hierbei muß man zwischen Design- und Laufzeitumgebung unterscheiden. Die Designumgebung dient dazu, den Benutzer beim Entwurf eines Sichtschemas zu unterstützen und daraus dann die schemaabhängigen Teile der Sprachbindung zu erzeugen. Der erzeugte Code bildet dann zusammen mit weiteren, schemaunabhängigen Modulen die Laufzeitumgebung, in der die Anwendungsprogramme mit der Datenbank arbeiten. Das hier entworfene Sichtsystem bildet dabei die zentrale Komponente für die Integration verschiedener Datenbanken im IRO–DB-Projekt ([GGF*96], [FFK96], [RFFS95]). Bild 17 beschreibt die dabei verwendete Drei-Schichten-Architektur ([GFF97]).

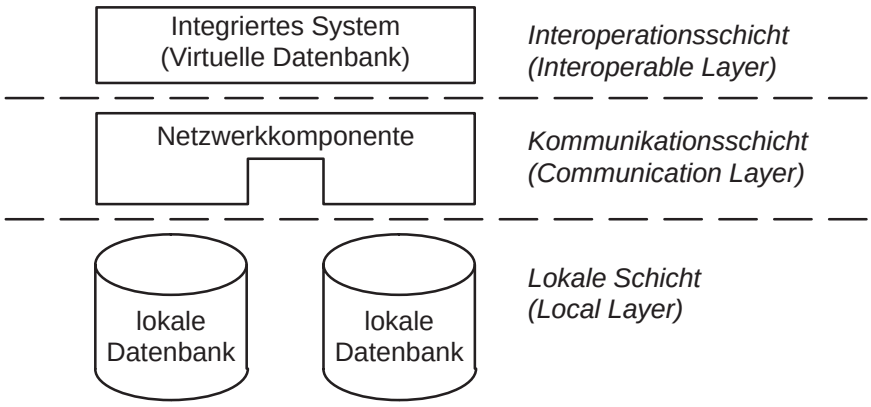


Bild 17: Gesamtarchitektur von IRO-DB

In der lokalen Schicht (*Local Layer*) liegen die heterogenen, zu integrierenden Datenbanken, die mit Hilfe von Adaptern (*Local Database Adapters, LDA*) eine ODMG–93-Schnittstelle zur Verfügung stellen. Im Rahmen von IRO–DB sind solche Adapter für die Datenbanksysteme Ontos, O_2, Matisse, Ingres und Sabrina entwickelt worden (siehe [D2–1/1] bis [D2–6/1]). Auf diese Adapter wird über eine Kommunikationsschicht (*Communication Layer*) zugegriffen, die OQL-Anfragen und Kommandos an die lokalen Datenbanken überträgt und Anfrageergebnisse und Objektstrukturen zurückliefert. Das Protokoll zum Objektzugriff (*Remote Object Access, ROA*) basiert auf einer objektorientierten Erweiterung von *Call Level Interface (CLI)* und *Remote Data Access (RDA)*. Auf der Kommunikationsschicht baut das integrierte System auf, das eine vollständige virtuelle Datenbank mit Objektmanagement, Metaschema, Anfragebearbeitung, Transaktionen usw. realisiert (*Interoperable Layer*).

4.6.1 Designumgebung

Die Designumgebung dient dazu, den Benutzer beim Entwurf eines Sichtschemas zu unterstützen und daraus dann die schemaspezifischen Teile der Laufzeitumgebung zu erzeugen. Für das eigentliche Design wird in IRO–DB ein graphisches Entwurfswerkzeug verwendet, mit dem eine deklarative Sichtdefinition erstellt wird. In einem nachfolgenden Schritt erzeugt ein Präprozessor hieraus die Sprachbindung für das Laufzeitsystem. Dieses Verfahren gliedert sich gut in die ODMG-Umgebung ein, da sie die Systemunabhängigkeit einer deklarativen Spezifikation und die nahtlose Einbettung in die Programmiersprache miteinander verbindet (vgl. Abschnitt 4.2, C++-ODL). Bild 18 zeigt den Aufbau der Designumgebung.

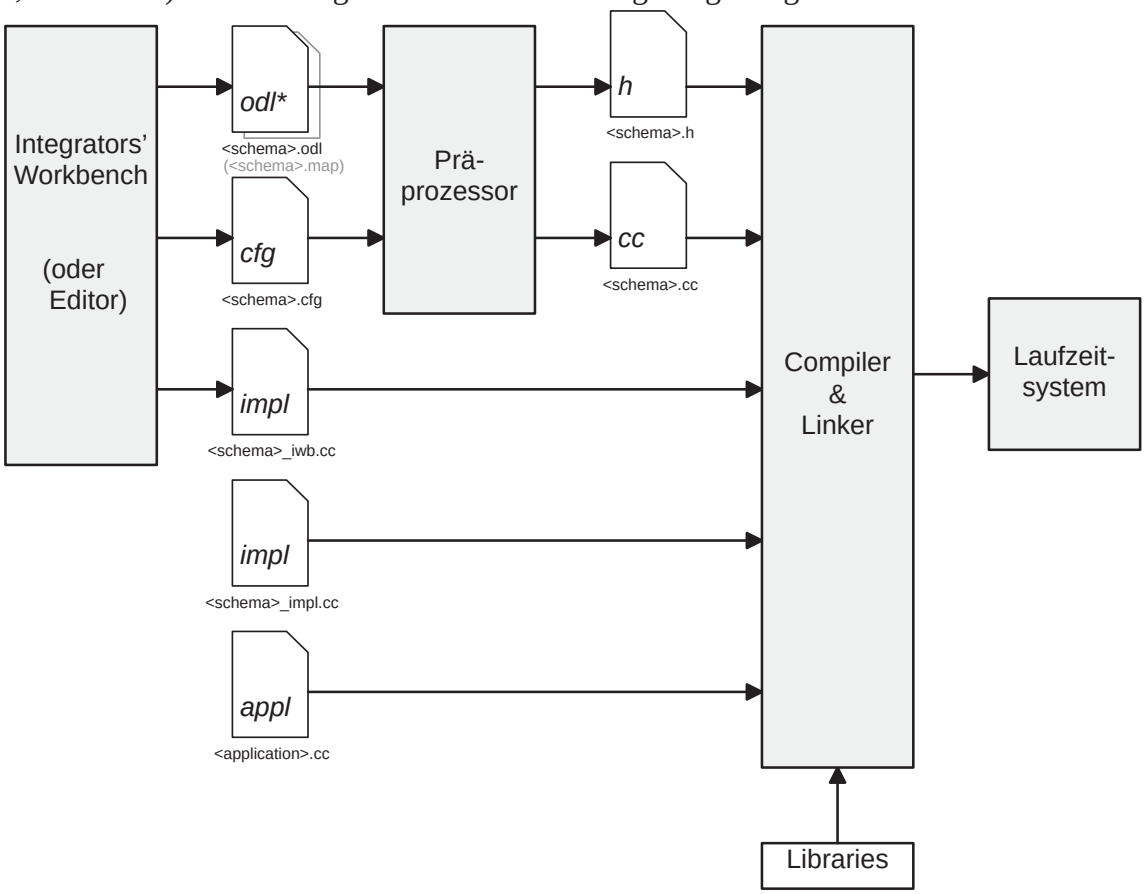


Bild 18: Aufbau der Designumgebung

Für die Erzeugung der Schemadateien verwenden wir die *Integrators' Workbench* ([D4–4/4], [BFHK94]). Sie bietet die Möglichkeit, ODL-Schemata zu importieren und graphisch darzustel-

len. Auf Basis einfacher Korrespondenzaussagen wird dann mit Hilfe einer Integrationsmethodologie ([D4–4/1], [Fan97]) ein integriertes Schema erstellt. Aus diesem Schema werden anschließend die deklarative Spezifikation und dazugehörige Methodenimplementierungen generiert. Statt mit der Workbench können diese Dateien aber auch mit einfachen Texteditoren oder anderen Entwurfsumgebungen erstellt werden.

Aus dieser deklarativen Sichtspezifikation erzeugt der Präprozessor die dazugehörige Sprachbindung. Die Datei <schema>.odl enthält die eingebettete Sichtspezifikation in ODL*-Syntax, wahlweise kann man auch eine reine ODL-Datei zusammen mit einer Mapping-Datei (<schema>.map) verwenden. Der Präprozessor liest diese Datei(en) und generiert daraus die Klassendefinitionen gemäß der Sprachbindung zusammen mit allen Hilfsfunktionen. Der erzeugte Code wird dabei in eine Header-Datei (<schema>.h) und eine Implementierungsdatei (<schema>.cc) aufgeteilt. Hierbei kann allerdings nicht die gesamte Sichtimplementierung automatisch aus der Schemadefinition erzeugt werden. Wir berücksichtigen deshalb noch zwei weitere Dateien, um die Implementierung zu vervollständigen:

<schema>_iwb.cc
: Diese Datei enthält Methodenimplementierungen, die sich aus dem Integrationsprozeß herleiten lassen. Hierbei handelt es sich um die Lese- und Schreiboperationen für virtuelle Datenfelder von Konvertierungsklassen, um die Konstruktoren zur Objekterzeugung, und um die spezifischen Relationshipklassen. Weiterhin können auch Hilfsroutinen zur Unterstützung dieser Operationen enthalten sein. Die Integration von Methoden wird von der Workbench noch nicht unterstützt, der hierfür generierte Code würde aber auch in diese Datei gehören. Ein Teil der hier enthaltenen Routinen könnte auch vom Präprozessor durch Analyse der Herleitungsklauseln in der ODL-Spezifikation erzeugt werden, die Erzeugung von der Workbench ist aber die einfachere Alternative, da die Herleitungsinformation dort bereits detailliert vorliegt. Außerdem ermöglicht dieser Ansatz, Lese- und Schreibmethoden asymmetrisch zu definieren, um Inkonsistenzen zu umgehen. Die Leseoperation zu einem Attribut könnte zum Beispiel immer nur auf die vertrauenswürdigste Datenbank zugreifen, während bei einem Schreibzugriff alle beteiligten Datenbanken verändert werden. Da mit ODL* nur reine Anfragefunktionalität ausgedrückt werden kann, kann eine solche variierte Schreibmethode nicht vom Präprozessor erzeugt werden.

<schema>_impl.cc
: Diese Datei enthält alle Methodenimplementierungen, die vom Benutzer hinzugefügt werden. Gemäß ODMG–93 werden Methoden nicht deklarativ, sondern direkt in der Programmiersprache implementiert. Die zugehörigen Methodensignaturen müssen allerdings in der ODL-Spezifikation enthalten sein.

Damit die zusätzlichen Methoden aus der IWB-Datei korrekt in der Header-Datei berücksichtigt werden können, übergibt die Workbench eine zusätzliche Konfigurationsdatei (<schema>.cfg) an den Präprozessor. Sie enthält Codeblöcke, die am Anfang oder Ende der Headerdatei, oder innerhalb einer Klassendeklaration eingefügt werden sollen. Die verwendete Syntax und ein Beispiel sind in Anhang F zu finden. Eine exakte, detaillierte Beschreibung aller Aufgaben des Präprozessors bei der Verarbeitung einer Schemadatei findet sich im Spezifikationsdokument [Bus96], die Implementierung anhand dieser Spezifikation wurde von Ibermática/Spanien übernommen.

4.6.2 Laufzeitumgebung

Bild 19 zeigt eine detailliertere Darstellung des integrierten Systems, das die eigentliche Laufzeitumgebung des Sichtsystems bildet. Die beiden Hauptelemente dieser Laufzeitumgebung,

die *Sprachbindung* und die *Objektverwaltung*, sind in den vorigen Abschnitten bereits beschrieben worden. Hinzu kommen die *Anfrageauswertung*, die *Transaktionsverwaltung* und ein *Sicherheitssystem* als Hauptmodule, das *Data Dictionary* zur Ablage der Schemainformation und eine *Programmierschnittstelle*, die die Sprachbindung um allgemeine Funktionen ergänzt. Zum Umfang dieser Arbeit gehören nur die Sprachbindung, die Objektverwaltung und die Programmierschnittstelle, da diese Komponenten vollständig von mir entworfen und implementiert worden sind. Das Design des Data Dictionary stammt ebenfalls von mir, implementiert worden ist es allerdings von Euriware/Frankreich. Die anderen Komponenten (Anfrageauswertung, Transaktionsverwaltung, Sicherheitssystem und Netzwerkkomponente) sind von anderen IPSI-Mitarbeitern bzw. unseren Partnern im IRO–DB-Projekt entworfen und implementiert worden (Universität Versailles und EDS/Frankreich, Ibermática/Spanien, FAW-Linz/Österreich). Die entsprechenden Schnittstellen zwischen den Modulen mußten allerdings in den eigenen Implementierungen berücksichtigt werden. Die folgenden Unterkapitel beschreiben kurz die einzelnen Module und ihr Zusammenspiel mit der Objektverwaltung.

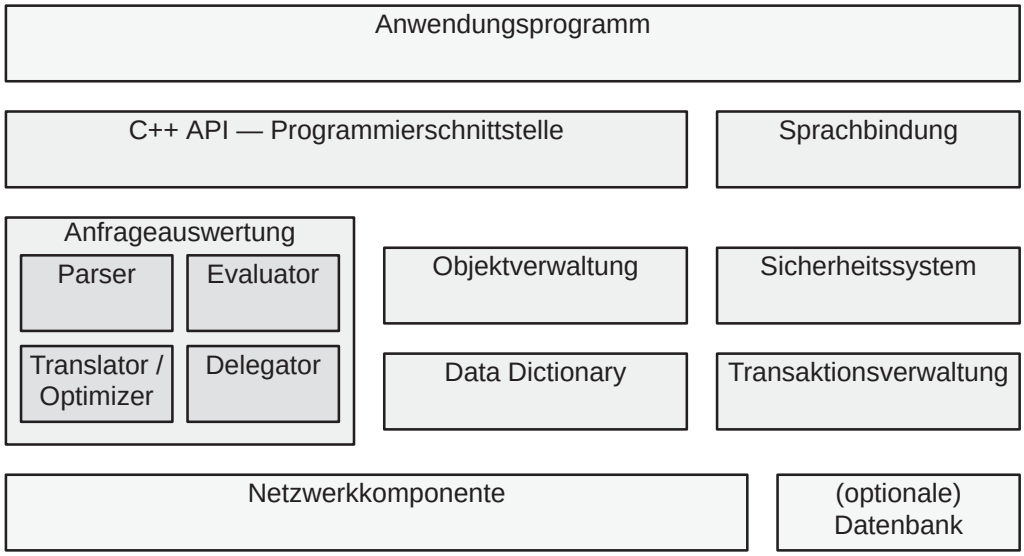


Bild 19: Vollständige Architektur der Laufzeitumgebung

4.6.2.1 Sprachbindung und Objektverwaltung

Diese beiden Module sind bereits beschrieben worden. Unter der Sprachbindung versteht man die schemaspezifischen Implementierungsklassen zusammen mit den drei Basisklassen Pobject, ImportClass und DerivedClass. Der entsprechende Code unterscheidet sich für jedes Schema und muß deshalb separat kompiliert und zum Anwendungsprogramm hinzugebunden werden. Die Objektverwaltung umfaßt alle Hilfsklassen zur Objekterzeugung und Referenzverwaltung sowie die Grundimplementierung für die Relationshipklassen. Um die Interaktion mit anderen Modulen zu vereinfachen, stehen drei Objektmanager OM, IOM und DOM zur Verfügung. Sie sind in Anhang H.2 beschrieben.

4.6.2.2 Anfrageauswertung

Die Anfrageauswertung dient der Bearbeitung deklarativer OQL-Anfragen. Der Parser analysiert die Anfrage und übersetzt sie in einen Operatorbaum; der Translator/Optimizer ersetzt Zugriffe auf Sichtkomponenten durch die entsprechenden OQL-Ausdrücke der Sichtdefinition und optimiert den Gesamtbaum für eine effiziente Auswertung. Der Evaluator übernimmt dann die eigentliche Auswertung, wobei Zugriffe auf externe Klassen über den Delegator an die Netz-

werkkomponente weitergereicht werden. Die Anfrageauswertung verwendet die beschriebenen Zugriffsmethoden der Objektverwaltung, um auf die Objekte in der Sprachbindung zuzugreifen. Die Objektverwaltung wiederum verwendet die Anfrageauswertung, um die Extentklauseln zu berechnen. Mehr Informationen zur Anfrageauswertung findet man in [FSF95] und [KaW97], Details zur Kostenabschätzung bei verteilter Anfrageauswertung in [GGT95] und [GST96].

4.6.2.3 Metaschema und Data Dictionary

Das Data Dictionary stellt die notwendigen Metainformationen für die Anfragebearbeitung zur Verfügung (siehe Kapitel 4.5.1 und Anhang H.4). Der Entwurf des Metaschemas stammt von mir, die Implementierung von Euriware/Frankreich.

4.6.2.4 Transaktionsverwaltung

Die Transaktionsverwaltung stellt auf Basis eines optimistischen Ticket-Verfahrens globale Transaktionen zur Verfügung ([GRS91]). Alle Datenbankzugriffe müssen innerhalb einer Transaktion erfolgen, und mittels eines Abhängigkeitsgraphen (*Global Serialization Graph, GSG*) kann durch Konflikterkennung die Serialisierbarkeit überprüft werden. Die Objektverwaltung muß vor einem Zugriff auf eine externe Datenbank sicherstellen, daß für diese Datenbank bereits eine Subtransaktion registriert worden ist. Bei Transaktionsende wird die Objektverwaltung über die betroffenen Objekte informiert, um mögliche Änderungen in die externen Datenbanken zurückzuschreiben. Die Zugriffsklasse für Transaktionen und die Implementierungen für globale und lokale Transaktionen sind im Anhang zu finden. Die Transkationsverwaltung ist in [TeW97], [TeW97a] und [D4-6/1] beschrieben, die Implementierung in [D4-6/2].

4.6.2.5 Sicherheitssystem

Das Sicherheitssystem dient der Modellierung von Datenbankbenutzern und deren Zugriffsrechten. Vor der Ausführung einer Anfrage oder eines Objektzugriffs muß überprüft werden, ob der aktuelle Benutzer die Berechtigung für diese Operation besitzt. Der Entwurf des Sicherheitssystems muß dabei die Besonderheiten des objektorientierten Modells berücksichtigen. Hierzu gehören etwa die Vererbung, oder die Möglichkeit, daß eine Methode, deren Ausführung dem Benutzer erlaubt ist, selbst andere Methoden aufrufen darf, die vom Benutzer nicht direkt ausgeführt werden dürfen. Für das Zusammenspiel mit dem Sicherheitssystem definiert der Objektmanager eine Reihe von Befehlen, die die aktuellen Berechtigungen überprüfen und im Falle eines unberechtigten Zugriffs sofort die aktuelle Transaktion abbrechen. Die Funktion om_assertWritePerm überprüft zum Beispiel die Schreibberechtigung und wird automatisch innerhalb von mark_modified aufgerufen. om_assertCreatePerm überprüft die Berechtigung zur Erzeugung neuer Objekte und wird innerhalb des Pobject-Konstruktors sowie in allen create_object-Methoden der Sprachbindung aufgerufen. Mit om_assertMethodPerm werden Methodenaufrufe und Lesezugriffe überprüft. Dies erfolgt jeweils in einer speziellen enterMethod-Methode, die vom Sicherheitssystem zur Verfolgung der Aufrufhierarchie gefordert wird. Das vollständige Sicherheitssystem ist in [EKP*96] und [EKPT96] beschrieben.

4.6.2.6 Netzwerkkomponente und Datenbank

Die Netzwerkkomponente (*Communication Layer*) ist bereits erwähnt worden. Sie stellt die Schnittstelle zu externen Datenbanken zur Verfügung. In IRO–DB ist das Sichtsystem als sogenannte *Middleware*, also als eigenständiges Programmsystem entwickelt worden, das zwischen dem Anwendungsprogramm und der Datenbank eingefügt wird. Es handelt sich also um ein rei-

nes Hauptspeichersystem, das selbst keine persistente Datenhaltung in einer Datenbank verwendet. Das Sichtsystem kann aber auch in eine Datenbank integriert werden, indem Funktionalität des Datenbanksystems für das Sichtsystem ausgenutzt wird, und die persistenten Objekte direkt über eine Datenbank-Objektverwaltung und nicht über die Netzkomponente angesprochen werden (siehe Seite 87). Der einfachste Fall ist hierbei, die Datenbank-Objektverwaltung nur als Ersatz für die Netzkomponente anzusehen und das Sichtsystem als eigenständige Komponente außerhalb der Datenbank zu belassen. Will man eine engere Kopplung, so müssen die in der Datenbank vorhandenen Module zur Anfrageauswertung und Transaktionsverwaltung sowie das Data Dictionary und das Sicherheitssystem um die notwendige Sichtfunktionalität erweitert werden bzw. mit den entsprechenden Modulen des Sichtsystems zusammenarbeiten können.

4.6.2.7 Programmierschnittstelle

Die Programmierschnittstelle dient dazu, Anwendungsprogrammen einen möglichst einfachen Zugriff auf das Datenbank- und Sichtsystem zu ermöglichen. Sie definiert vier Teile:

i) die ODMG-Klasse Database für die Interaktion mit der Datenbank

ii) die ODMG-Klasse Transaction zum Starten und Beenden globaler Transaktionen

iii) eine Reihe von oql-Funktionen zur Ausführung deklarativer Datenbankanfragen

iv) eine globale Klasse IRO, die vereinfachte Grundfunktionen zur Verfügung stellt

Eine Beschreibung der einzelnen Teile sowie die konkreten Klassendefinition sind in Anhang H.7 zu finden. Die Spezifikation der Programmierschnittstelle ist in [D5-1/2] gegeben.

Beispiel 4.9: *Programmbeispiel*

Ein typisches Anwendungsprogramm, das die gerade beschriebene Programmierschnittstelle verwendet, hat grundsätzlich den im folgenden Code-Beispiel verwendeten Aufbau. Innerhalb der Transaktion wird hier direkt mit der Sprachbindung gearbeitet, um die Anzahl der Veröffentlichungen aller Autoren auszugeben:

```
int main () {
    // Initialisierung
    Database* theDB;
    IRO::initialize ();
    theDB = IRO::openDatabase (”dbname”);
    theDB→set_user_pwd (”globaluser”,”globalpassword”);
    theDB→set_user_pwd (”ldaid”, ”localuser”, ”localpassword”);
    ...

    // Datenbankzugriffe innerhalb einer globalen Transaktion
    IRO::beginTransaction ();
        Set < Ref <Author> > authors = Author::get_extent ();
        Iterator < Ref <Author> > it = authors.create_iterator ();
        while ( it.more () ) {
            cout << ”Author ” << it.value ()→fname () << ” ” << it.value ()→lname ()
                 << ” has ” << it.value ()→publications ()→cardinality () << ” publications.” << endl;
            it.next (); };
    IRO::commitTransaction ();

    // Programmende
    IRO::closeDatabase ();
    IRO::shutdown ();
    return 0;
    };
```

■

5 Konkurrierende Updates von Sichten

Der bisherige Teil der Arbeit hat sich immer nur mit den Sichten eines einzelnen Anwenders befaßt. In diesem Kapitel soll nun beschrieben werden, wie die gemeinsame, konkurrierende Arbeit mehrerer Anwender auf einem Datenbanksystem unterstützt wird. Außerdem wird noch genauer auf die Probleme eingegangen, die bei Updates auftreten.

5.1 Prozeßmodell und Sperren

Um den gegenseitigen Einfluß mehrerer Benutzer beurteilen zu können, muß man deren Arbeitsbereiche kennen, also die Daten, auf die der jeweilige Benutzer direkten Zugriff hat. In einem traditionellen Datenbanksystem kann man davon ausgehen, daß die Datenbank als Server einen logischen Prozeß darstellt, auf den alle Benutzer zugreifen, und jeder Benutzer zusätzlich in seinem Anwendungsprogramm (Client) einen eigenen Datenbereich besitzt, auf den nur er Zugriff hat (siehe Bild 20).

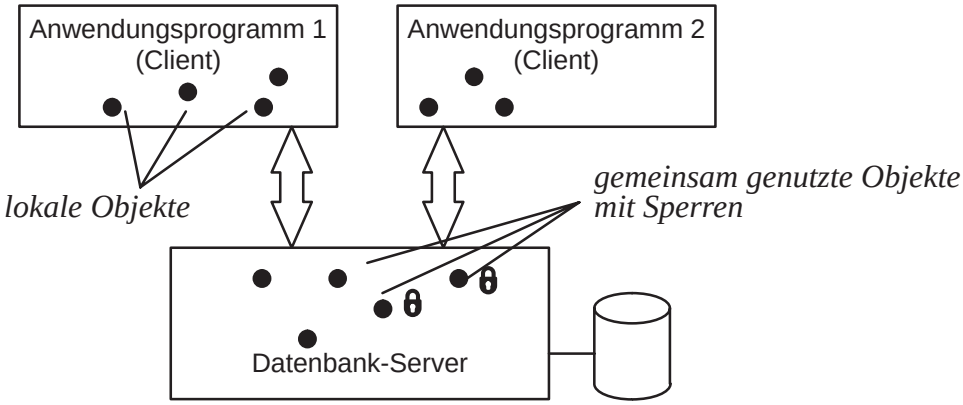


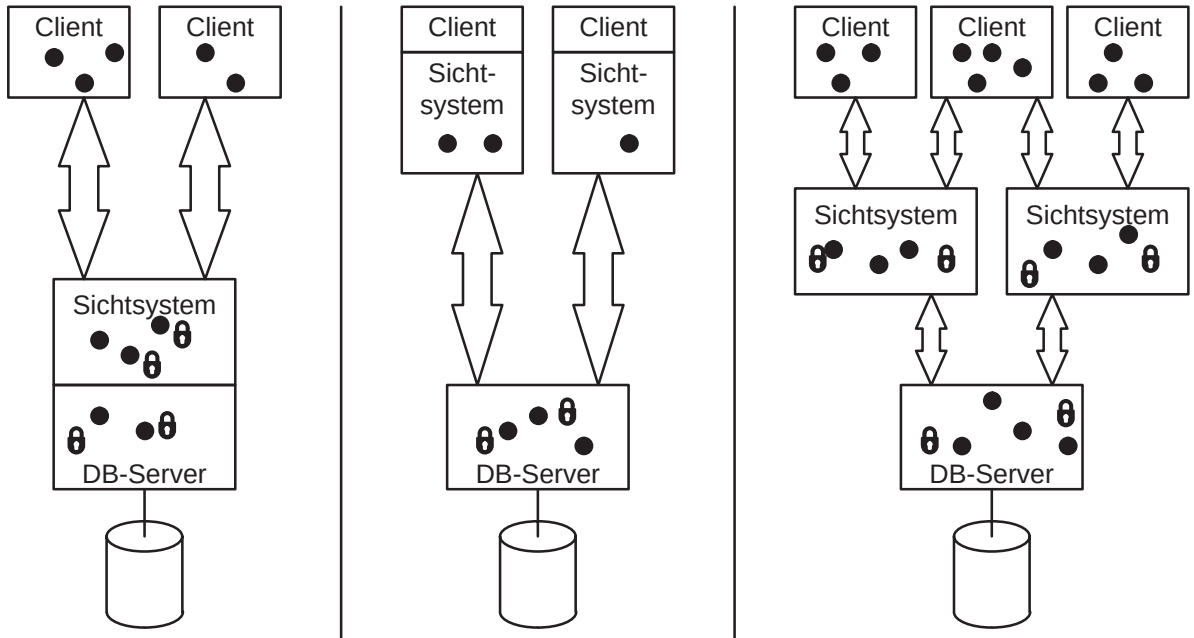
Bild 20: Datenbank-Client-Server-Architektur

Die Datenverwaltung im Anwendungsprogramm bleibt vollständig unter dessen Kontrolle, da es von außen keine Zugriffsmöglichkeiten gibt. Die Daten in der Datenbank werden jedoch gemeinsam genutzt, und die Zugriffe müssen deshalb mittels Transaktionen koordiniert werden, damit jeder Benutzer immer einen konsistenten Datenbestand zur Verfügung hat (ACID-Eigenschaft: Atomizität, Konsistenz, Unabhängigkeit, Dauerhaftigkeit[27]). ODMG–93 verwendet hierzu standardmäßig ein Sperrverfahren, bei dem Objekte, auf die zugriffen wird, implizit oder explizit mit einer Sperre versehen werden, um anderen Benutzern gegebenenfalls den Zugriff auf inkonsistente Zwischenzustände zu verwehren. Der Standard definiert dabei nur relativ allgemeine Forderungen an das Transaktionssystem, so daß man statt des strikten, pessimistischen Sperrverfahrens auch andere Transaktionskonzepte verwenden kann, sofern sie ein vergleichbares Verhalten aufweisen ([TeW97a]). Für die weiteren Betrachtungen sei aber das Sperrverfahren angenommen.

Fügt man das Sichtsystem in die Architektur aus Bild 20 ein, so ergeben sich prinzipiell drei mögliche Prozeßmodelle, die in Bild 21 dargestellt sind. In Fall a) wird das Sichtsystem vollständig in die Datenbank integriert. Da in diesem Fall alle Sichtobjekte von allen Anwendern gemeinsam

27. Zu Datenbankeigenschaften, Transaktionen und Sperrverfahren sei auf die Standard-Datenbankliteratur verwiesen.

genutzt werden, müssen hierfür geeignete Sperren zur Verfügung gestellt werden. In Fall b) wird jedem Anwender ein eigenes Sichtsystem zur Verfügung gestellt. Die Sichtobjekte können nur von dem jeweiligen Anwender verwendet werden, so daß Sperren auf dieser Ebene nicht notwendig sind. Die Koordination der einzelnen Benutzer erfolgt wie vorher nur durch Sperren auf der Serverseite. Fall c) zeigt ein verallgemeinertes Modell der beiden vorigen Verfahren, in dem jede Sicht als eigenständiger Vermittlerprozeß zwischen Anwender und Datenbank auftritt. Eine solche Architektur findet man zum Beispiel bei der Übertragung des Ansatzes auf CORBA (Abschnitt 6.7). In diesem Fall verwendet man Sperren innerhalb der Sicht, um die Zugriffe der Benutzer auf die Sichtobjekte zu koordinieren, und zusätzliche Sperren in der Datenbank, um die Zugriffe der einzelnen Sichten auf die Originalobjekte voneinander zu trennen.



a) Sicht im Server *b) Sicht im Client* *c) Sicht als Middleware*

Bild 21: Prozeßmodelle mit eingefügtem Sichtsystem

Die folgenden Untersuchungen zu konkurrierenden Updates beziehen sich auf Variante b), da diese am einfachsten zu realisieren ist und im IRO–DB-Prototyp angewandt wurde. Hierbei sind im Sichtsystem keine Sperren notwendig, da alle Manipulationen innerhalb derselben Transaktion stattfinden müssen. Transaktionen selbst sind auch im Sichtsystem notwendig, um die zugehörigen externen Transaktionen auf der Datenbank zu gruppieren und die Gültigkeit von Objekten bestimmen zu können.[28]

5.2 Lokale Updates

Lokale Updates sind jene, die von Benutzern ausgeführt werden, die direkt mit der externen (lokalen) Datenbank arbeiten (siehe Bild 22). Aus Sicht der Datenbank handelt es sich beim Sichtsystem ebenfalls nur um eine normale Anwendung eines Benutzers, so daß konkurrierende Updates von Sichtbenutzern und lokalen Benutzern gleichbedeutend sind mit konkurrierenden

28. Objektidentität muß von den lokalen Datenbanken nur innerhalb einer Transaktion zugesichert werden. Nach Ende einer Transaktion sind somit alle virtuellen Objekte als ungültig zu markieren, da sich bei einer nachfolgenden Transaktion die OIDs der zugehörigen Basisobjekte geändert haben können.

Updates zweier lokaler Benutzer. Die Konsistenzerhaltung kann deshalb dem Transaktionssystem der externen Datenbank überlassen werden.

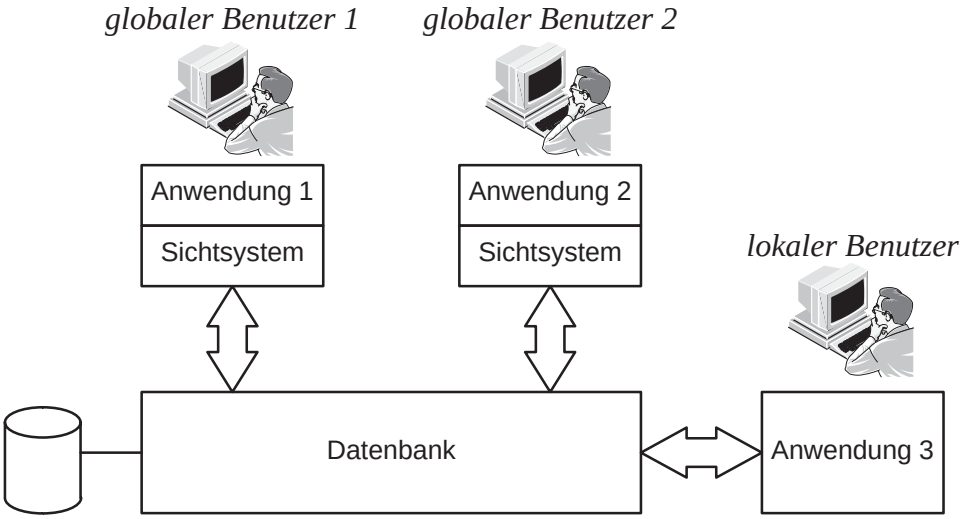


Bild 22: Lokale und globale Benutzer

5.3 Fremde Updates

Das im vorigen Abschnitt gesagte gilt auch für zwei unterschiedliche Benutzer des Sichtsystems. Da jeder Benutzer seine eigenen Sichtobjekte verwendet, kommt es auf dieser Ebene zu keinen Konflikten. Der gemeinsame Zugriff auf die Originalobjekte in der Datenbank wird von dem dort vorhandenen Transaktionssystem gesteuert. Jeder Sichtbenutzer kann wie ein lokaler Benutzer behandelt werden, so daß die Konsistenz ohne weitere Maßnahmen zugesichert werden kann. Man muß jedoch darauf achten, daß nicht nur die einzelnen Datenbankobjekte durch Sperren koordiniert werden, sondern auch persistente Objektmengen, wie zum Beispiel die Extents. Erzeugt ein Benutzer ein neues Objekt, so wird dabei auch der Extent der entsprechenden Klasse vergrößert. Ohne eine Sperre auf dem Extent wäre diese Änderung für andere Benutzer bereits innerhalb ihrer Transaktion sichtbar, was zu Fehlern bei der Auswertung der Extentklauseln der virtuellen Klassen führen kann. Zu Sperrverfahren auf objektorientierten Datenmodellen gibt es bereits Literatur, die die notwendige Granularität (Attribute, Objekte, Klassen), das Sperren ganzer Pfade (multi-level locks) und die Minimierung unnötiger Sperrkonflikte (intention locks, semantische Sperren, implizite Sperren) untersucht.

Weiterführende Literatur

Die folgende Liste nennt einige untersuchte Veröffentlichungen zu den einzelnen Themen: predicate locks [EGLT76], granularity locks/intention locks (MGL) [GLPT76], hierarchical locks [SiK80], multilevel locks [Lyn83] [Wei86], dynamic directed graph synchronization (DDG) [CHM92], object-oriented locking [AEA92] [HaH91] [HeW88], open nested transactions [MRW*93], locks using abstraction relationships' semantics (LARS) [FRH97]. Der letztgenannte Artikel enthält darüber hinaus eine Übersicht über die Entwicklung der verschiedenen Sperrverfahren sowie die konkreten Transaktionsmechanismen von Orion, O_2, GemStone und ObjectStore.

5.4 Eigene Updates

Als letzter Fall bleibt die Synchronisation von Updates eines einzelnen Benutzers. Im Gegensatz zu den vorigen Fällen, wo die Änderungen eines Benutzers vor einem anderen versteckt werden

müssen, um während der Zugriffszeit einen konsistenten Datenbestand zuzusichern, geht es hierbei darum, die Änderungen sichtbar zu machen, also konsistent in allen abhängigen Objekten durchzuführen. Diese Abhängigkeiten ergeben sich sowohl bei Importobjekten als auch bei Konvertierungsobjekten, allerdings in unterschiedlicher Ausprägung. Im ersten Fall muß man die Konsistenz zwischen der externen Datenbank und den in den Importobjekten zwischengespeicherten Werten beachten, im zweiten Fall die Gültigkeit und Identität der abgeleiteten Objekte.

Für eine detailliertere Untersuchung muß man zwischen zwei Formen von Updates unterscheiden: Objektänderung und Extentänderung.

Definition 5.1: Objektänderung, Extentänderung

Ein Update, das den Zustand eines Objekts ändert, also Attributwerten oder Relationships neue Werte zuweist, heißt *Objektänderung*. Ein Update, das vorhandene Objekte löscht oder neue Objekte erzeugt, heißt *Extentänderung*. Eine einzelne Update-Operation kann sowohl Objektänderungen als auch Extentänderungen verursachen.

Durch Kombination ergeben sich die folgenden acht Abhängigkeiten, der Pfeil beschreibt dabei den Datenfluß:

1) Importobjekt → externe Änderung (Objekt)

Importobjekte verwenden eine Caching-Strategie, so daß Änderungen nicht sofort in die externe Datenbank zurückgeschrieben werden. In bestimmten Situationen ist jedoch ein Zurückschreiben zwingend erforderlich. Der erste Fall ist das Persistentmachen der Daten beim Deaktivieren eines geänderten Importobjekts, sowohl innerhalb als auch am Ende einer Transaktion. Bei der Deaktivierung im Verlauf einer Objektlöschung kann hingegen auf das Zurückschreiben verzichtet werden, da das externe Objekt sowieso gelöscht wird. Der zweite Fall ist der direkte Zugriff auf die externen Daten durch denselben globalen Benutzer. Dies geschieht in drei Situationen:

a) Beim Aktivieren eines Importobjekts

   Hier kann es zu keinen Konflikten kommen, da das Objekt noch nicht aktiviert ist und somit keine Änderungen enthalten kann. Auf fremde Objekte wird bei der Aktivierung nicht zugegriffen.

b) Beim Ausführen von Methoden

   In meinem System gehe ich davon aus, daß die in der externen Datenbank definierten Methoden auch dort ausgeführt werden müssen (siehe Abschnitt 3.3.4). Da diese Methoden direkt auf den externen Daten arbeiten, müssen vor einer Methodenausführung *alle* Objektänderungen in diese Datenbank zurückgeschrieben werden. Es reicht nicht aus, nur das Objekt zurückzuschreiben, das die Methode ausführt, da innerhalb der Methode auf jedes Element der Datenbank zugegriffen werden kann. Änderungen, die sich auf andere externe Datenbanken beziehen, sind hiervon nicht betroffen.

c) Beim Ausführen von Anfragen

   Genau wie Methoden greifen auch Anfragen direkt auf die Werte des externen Systems zu. Auch hier müssen daher die Änderungen vor der Auswertung zurückgeschrieben werden.

2) externe Änderung → Importobjekt (Objekt)

Die Änderung an einem externen Objekt muß in der Sicht sichtbar gemacht werden. Eine solche Änderung kann aufgrund der Transaktionentrennung nicht von einem externen Benutzer stam-

men. Auch die Änderungen, die in Fall 1) in die Datenbank zurückgeschrieben worden sind, sind ohne Bedeutung, da die entsprechenden Änderungen bereits in der Sicht vorliegen. Problematisch sind hingegen Änderungen, die durch Anfragen oder Methodenausführungen auf dem externen System hervorgerufen werden. Hierbei muß ermittelt werden, welche Änderungen vorgenommen worden sind, um die Sicht entsprechend zu aktualisieren. Auf diesen Problemfall wird im Anschluß gesondert eingegangen (siehe Abschnitt 5.5).

3) Importobjekt → externe Änderung (Extent)

Dieser Fall kann nicht auftreten, da jede Erzeugung eines Importobjekts so formuliert ist, daß zuerst das externe Objekt erzeugt und dieses dann importiert wird.

4) externe Änderung → Importobjekt (Extent)

Genau wie in Fall 2) kann das Erzeugen oder Löschen externer Objekte nur durch die Ausführung von Anfragen oder Methoden verursacht werden. Die hierbei auftretenden Probleme werden in Abschnitt 5.5 behandelt.

5) Konvertierungsobjekt → Basisobjekt (Objekt)

Konvertierungsobjekte sind so definiert, daß die Zugriffsmethoden direkt auf die Basisobjekte zugreifen. Jede Änderung eines Konvertierungsobjekts ist also direkt im zugehörigen Basisobjekt sichtbar.

6) Basisobjekt → Konvertierungsobjekt (Objekt)

Genau wie in Fall 5) greifen Konvertierungsobjekte direkt auf die Daten der Basisobjekte zu. Jede Änderung eines Basisobjekts wird also automatisch sichtbar. Problematisch ist jedoch, wenn eine Objektänderung eines Basisobjekts zu einer Extentänderung in der Konvertierungsklasse führt. Dies kann zum Beispiel dann auftreten, wenn ein Attribut verändert wird, das in der Extentklausel der Konvertierungsklasse als Join- oder Selektionsprädikat verwendet wird. Dieser Fall ist gleichbedeutend mit den in relationalen Sichten bekannten Update-Anomalien (siehe Seite 20). Für ein konsistentes Systemverhalten muß man solche Änderungen generell ausschließen, jeder Schreibzugriff auf solch ein *kritisches* Datenfeld muß also systemseitig unterbunden werden. Sind solche Updates im System dennoch erwünscht — einschließlich aller dabei auftretenden Seiteneffekte —, kann man die zugehörige Schreibmethode so gestalten, daß alle notwendigen Maßnahmen zur Konsistenzerhaltung von dieser Methode ausgeführt werden. Die Erkennung und Unterstützung solcher Updates wird vom implementierten System nicht zur Verfügung gestellt.

7) Konvertierungsobjekt → Basisobjekt (Extent)

Dieser Fall kann, genau wie 3), nicht auftreten, da die Erzeugung von Konvertierungsobjekten immer auf der vorhergehenden Erzeugung (oder Veränderung) von Basisobjekten beruht. Die Löschoperation von Konvertierungsklassen ist ebenfalls dafür zuständig, die entsprechenden Änderungen an den Basisobjekten vorzunehmen. Eine weitergehende Unterstützung durch das System ist nicht erforderlich.

8) Basisobjekt → Konvertierungsobjekt (Extent)

Im Gegensatz zu Objektänderungen werden Extentänderungen von Basisklassen nicht automatisch in den Konvertierungsklassen sichtbar, sondern erst bei der nächsten Auswertung der Ex-

tentklausel. Wird auf den Extent zugegriffen, kann die Extentklausel überprüft und gegebenenfalls neu ausgewertet werden. Bei teilweiser Instanziierung geschieht dies sowieso, sobald die Klasse vollständig instanziiert ist, kann man die erneute Auswertung aus Effizienzgründen durch eine Markierung verhindern. Bei einer Extentänderung in den Basisklassen muß diese Markierung dann aber zurückgesetzt werden, um eine Neuauswertung der Klausel beim nächsten Extentzugriff anzustoßen. Die Extentklausel wird jedoch nicht ausgewertet, wenn mit bereits instanziierten Konvertierungsobjekten gearbeitet wird. Wird das zugrunde liegende Basisobjekt allerdings gelöscht, so muß sichergestellt werden, daß auch auf das Konvertierungsobjekt nicht mehr zugegriffen werden kann. Dies kann durch Überprüfung der Basisobjekte bei *jedem* Zugriff auf das Konvertierungsobjekt, durch Erzeugung eines Ausnahmesignals bei Zugriff auf ein ungültiges Basisobjekt, oder durch explizite Notifikation der Konvertierungsklasse bei Extentänderung der Basisklasse erfolgen. Die notwendige Neuberechnung der Extentklauseln kann dabei durch Verwendung inkrementeller Anfrageauswertung optimiert werden.

Im Gegensatz zum relationalen Modell, bei dem die Konsistenzerhaltung abhängiger Daten und die Abbildung von Sicht-Updates auf Datenbank-Updates vom System erbracht werden muß, erlaubt die objektorientierte Modellierung mit Hilfe von Methoden eine explizite, situationsabhängige Spezifikation aller notwendigen Operationen. Dies erfordert zwar einigen Aufwand, erlaubt aber auch dann die Ausführung von Updates, wenn die notwendigen Operationen nicht mehr vom System bestimmt werden können.

5.5 Auswertung externer Methoden und Anfragen

Ein Hauptproblem bei der Konsistenzerhaltung des Sichtsystems ist die Ausführung von Methoden und Anfragen in der externen Datenbank. Durch den direkten Zugriff auf die externen Daten müssen vor jeder solchen Auswertung alle Änderungen an den Importobjekten in die externe Datenbank zurückgeschrieben werden. Viel schwerwiegender ist jedoch, daß bei der Ausführung prinzipiell jedes Datenbankobjekt beliebig verändert werden kann — nach einer Anfrage- oder Methodenausführung muß also der gesamte Datenbestand der Sicht mit dem aktuellen Zustand der externen Datenbank abgeglichen werden. Im Falle von Anfragen kann man möglicherweise entscheiden, ob überhaupt Schreiboperationen vorgenommen werden, und welche Objekte oder Klassen davon gegebenenfalls betroffen sind. Bei Methoden jedoch hat man keinerlei Information darüber, was für Seiteneffekte auftreten können.

Bei der Prototyp-Implementierung wird daher bei jeder Methodenausführung der gesamte Datenbestand der Sicht als ungültig markiert, was zu einer automatischen Neuberechnung der Extents beim nächsten Zugriff auf die Sicht führt. Dieses Verhalten führt zu einem extremen Performanzverlust des Systems, so daß dieser Ansatz bei großen Datenmengen und häufiger Methodenausführung auf den externen Objekten praktisch nicht anwendbar ist. Ein erster Schritt zur Verbesserung ist die Optimierung der Extentauswertung. Kennt man den alten und den neuen Extent sowie die Auswirkung möglicher Änderungen auf die Importklassen, so kann möglicherweise eine Differenzanfrage formuliert werden, die die gelöschten, hinzugefügten und veränderten Objekte bestimmt und die notwendigen Korrekturen ohne vollständige Neuberechnung des Extents vornimmt.

Eine effiziente Verbesserung des Systems kann allerdings nur durch Verwendung von mehr Metainformation oder durch aktive Unterstützung durch das externe System herbeigeführt werden. In VODAK ([Kla93]) können Methoden zum Beispiel als READONLY, in C++ als const markiert werden. Hierdurch wird zugesichert, daß keinerlei (sichtbare) Veränderungen an den Objekten vorgenommen werden. In einem solchen Fall kann auf die Invalidierung der Importobjekte ver-

zichtet werden. Für Methoden, die Updates vornehmen, könnte man die Spezifikation um Update-Indikatoren erweitern oder die Methode selbst analysieren, um die potentiell veränderten Objekte eingrenzen zu können. Methoden, die nur das Objekt selbst verändern, könnten etwa mit LOCAL, Methoden, die nur Objekte bestimmter Klassen verändern, mit UPDATES CLASS1, CLASS2 markiert werden[29]. Neben der Verfolgung von Veränderungen kann die Klassifikation von Methoden in lesende und schreibende auch die Parallelität von Transaktionen verbessern ([CaF92], [Rak97]).

Eine aktive Unterstützung zur Laufzeit könnte zum Beispiel dadurch zur Verfügung gestellt werden, daß jede Methode nach ihrer Ausführung die Menge der von ihr geänderten Objekte zurückliefert (*change set*). Die Importobjektverwaltung kann dann gezielt auf die Änderungen reagieren. Eine andere Möglichkeit ist die aktive (asynchrone) Meldung geänderter Objekte vom externen System an das Sichtsystem; zum Beispiel mit Hilfe der Modifikationslisten von [AlP96], [AlE99]. Jede externe Objektänderung kann so erkannt und das zugehörige Importobjekt angepaßt werden. Eine weitere Variante ist die Verwendung von Versions-Zeitstempeln, die Objektveränderungen in der Datenbank dokumentieren (vgl. [ABF02]). Diese erfordern jedoch immer noch eine Überprüfung jeder einzelnen Instanz, die dann jedoch schneller durchgeführt werden kann. Grundsätzlich ist einem möglichst großen statischen Vorwissen der Vorzug gegenüber rein dynamischen Verfahren zu geben, da jede Überprüfung eines externen Objekts mit zeitaufwendigen Netzwerkzugriffen verbunden ist.

Weiterführende Literatur

Ein Basiswerk für inkrementelle Updates materialisierter (relationaler) Sichten ist von Blakeley et al. ([BLT86]); Qian und Wiederhold verallgemeinern das Konzept für andere Datenbankausdrücke, wie etwa Trigger, aktive Anfragen und Monitore ([QiW91]). Roussopoulos gibt eine Kostenanalyse für die verschiedenen Verfahren ([Rou91]), und Bækgaard und Mark stellen eine Erweiterung vor, um die inkrementelle Anfrageberechnung auch für genestete Anfragen nutzen zu können ([BaM95]). Ceri et al. untersuchen inkrementelle und verzögerte Updates bei replizierten Daten ([CHKS92]), Transaktionen in Multidatenbank-Systemen werden von Breitbart et al. beschrieben ([BGS95]). Stonebraker führt ein Regelsystem in POSTGRES ein, um Sichten besser unterstützen zu können ([SJGP90]), Widom et al. verwenden ebenfalls regelbasierte Trigger zur automatischen Konsistenzerhaltung abhängiger Daten ([WiF90], [BaW95]). Ein Updateverfahren für strukturelle Objektsichten auf relationalen Datenbanken wird von Barsalou et al. vorgestellt ([BKSW91]). Caching-Verfahren für objektorientierte Datenbanksysteme werden zum Beispiel von Franklin behandelt ([Fra96]). Das Problem der Seiteneffekte bei Methodenausführung tritt unter anderem auch bei der Konsistenzerhaltung methodenbasierter Indizes auf (siehe [Zdo93], S. 18).

29. OSCAR ([Kla92]) verwendet bei der Methodendeklaration die Schlüsselworte USES und MODIFIES, um einem Optimierer bekanntzugeben, welche Daten gelesen bzw. verändert werden.

6 Anwendung auf andere Systeme

Dieses Kapitel befaßt sich damit, wie das für ODMG–93 und C++ vorgestellte Sichtkonzept auf andere Systemumgebungen anwendbar ist. Als alternative Implementierungssprache ist die C++-nahe Programmiersprache Java von aktuellem Interesse. Anschließend werden die neuen Versionen von ODMG–93 untersucht, die im Entwurf nicht berücksichtigt werden konnten. Für relationale und objektrelationale Datenbanken ist das Sichtsystem wenig geeignet, an andere Objektumgebungen ist es hingegen anpaßbar. Eine detaillierte Beschreibung einer möglichen Übertragung der Konzepte auf die verteilte Objektumgebung CORBA schließt das Kapitel.

6.1 Anpassung an Java

Die Programmiersprache Java wurde als möglichst einfache objektorientierte Programmiersprache konzipiert, die plattformunabhängig sichere und verteilte Anwendungen ermöglicht:

> *"Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language." [Fla97, S.3]*

Innerhalb kürzester Zeit entwickelte sie sich zur Internet-Programmiersprache Nummer eins und wird immer mehr auch zur Anwendungsentwicklung verwendet, bei der die Plattformunabhängigkeit und die Verfügbarkeit einer graphischen Oberfläche ausgenutzt werden können. Syntaktisch basiert Java auf C++; es erlaubt aber keine maschinennahen Strukturen und Operationen und verwendet ein anderes Ausführungsmodell. Die komplexe Mehrfachvererbung ist zum Beispiel durch die Definition funktionaler Schnittstellen mittels *Interfaces* ersetzt worden. Zeiger und Funktionszeiger stehen nicht mehr zur Verfügung, wodurch direkte Speichermanipulationen ausgeschlossen werden; außerdem übernimmt Java mit seiner *Garbage Collection* selbst das Löschen nicht mehr benötigter Objekte, so daß Fehler durch vorzeitiges Löschen und Zugriff auf nicht mehr gültige Speicherbereiche nicht mehr auftreten können. Zur Ausführung eines Java-Programms wird ein Interpreter verwendet, der den aus dem Quelltext erzeugten Bytecode interpretiert. Der Interpreter ist in der Lage, zur Laufzeit die Abhängigkeit von Programmteilen zu erkennen, Klassen dynamisch einzubinden und sie bei Bedarf sogar in Bytecode zu übersetzen.

Ab ODMG–2.0 existiert auch eine Sprachbindung für Java ([CaB97]). Analog zur C++-Sprachbindung werden die Literale in primitive Datentypen und die ODL-Klassendefinitionen in Java-Klassen übersetzt. Es handelt sich also auch hier um eine einbettende Abbildung. Extentverwaltung und Relationship-Implementierungen sind in der Sprachbindung nicht definiert und die Persistenz soll dynamischer gehandhabt werden als in C++ (*persistence by reachability*).

Auf Grund der Ähnlichkeit der Sprachen und der Sprachbindungen kann das vorgestellte Sichtkonzept auch auf Datenbanken auf Basis von Java angewandt werden. Auch in Java ist mehrfache Klassenzugehörigkeit eines Objekts nicht möglich, so daß auch hier Stellvertreterobjekte in neuen Klassen angelegt werden müssen, wenn andere Strukturen oder Funktionalitäten eines Objekts gewünscht werden. Ein Objekt kann zwar mehrere Interfaces zur Verfügung stellen, die Implementierung der jeweiligen Funktionen muß aber in der Klasse geschehen — das Hinzufügen eines Interfaces erfordert somit, analog zu C++, das Verändern der Klasse. Die wesentlichen Unterschiede bei der Behandlung von C++ und Java sind in [CaB97] zu finden. Diese müssen, genau wie die Java-Syntax und das Java-Datenmodell, bei der Übertragung des Sichtkonzepts beachtet werden. Bedeutsam sind dabei vor allem die folgenden Unterschiede:

- Keine Zeiger:
  Java stellt zwar keine expliziten Zeiger zur Verfügung, die Objektreferenzen basieren intern aber dennoch auf einem Zeigerverfahren. Diese Funktionalität reicht für das Sichtsystem aus.
- Keine Funktionszeiger:
  Da es in Java keine Funktionszeiger gibt, muß die entsprechende Funktionalität durch Zugriffsobjekte zur Verfügung gestellt werden. Hierzu definiert man für jeden Funktionstyp eine generische Klasse (oder ein Interface) mit einer standardisierten Funktion, zum Beispiel eval. Für jede konkrete Funktion, die übergeben werden soll, implementiert man eine entsprechende Subklasse (bzw. eine Interface-Implementierung), die innerhalb von eval genau die gewünschte Funktion aufruft. Anstelle des Funktionszeigers wird dann eine Instanz dieser Klasse übergeben.
- Introspektion:
  Mit Hilfe des sogenannten *Reflection API* kann man Strukturinformationen zur Verfügung stellen und dynamisch Methoden aufrufen. Die Erzeugung von Wrappern ist daher nicht mehr unbedingt notwendig, kann aber zu einem Effizienzgewinn führen, da das dynamische Zusammenstellen von Funktionsaufrufen recht zeitaufwendig ist. Dasselbe gilt für die generischen Zugriffsfunktionen für Properties.
- Dynamische Übersetzung:
  Java erlaubt es, zur Laufzeit neue Klassen zu übersetzen und zum System hinzuzufügen. Hierdurch ist es möglich, auch während des Betriebs neue Sichten zu definieren, da ein Übersetzungsschritt für das Gesamtsystem nicht mehr notwendig ist.
- Sicherheitskonzept:
  Da Java selbst Funktionen zur Vergabe und Überprüfung von Benutzerberechtigungen zur Verfügung stellt, sollte man das Sicherheitsmodul der Laufzeitumgebung an diese Möglichkeiten anpassen.
- Objekte statt Templates:
  Java bietet keine Templates, mit denen die mengenwertigen Typkonstruktoren dargestellt werden könnten. Statt dessen verwendet man Mengenklassen, die auf generischen Objekten vom Typ Object operieren. Um Mengen von Literalen zu bilden, muß man statt der eigentlichen Literale (int, float, ...) die korrespondierenden Objekttypen (Integer, Float, ...) verwenden.

[CaB97] definiert keine Relationship-Funktionalität in der Java-Sprachbindung. Die vorgestellte Relationship-Implementierung läßt sich nach Java übernehmen.

Ein wesentlicher Unterschied zu C++ kann sich aus dem normierten Bytecode ergeben, der zur Laufzeit interpretiert wird. Durch diese Normierung stellen alle Java-Systeme die gleiche Laufzeitumgebung zur Verfügung, so daß für Importobjekte möglicherweise ein dynamischer Codeaustausch verwendet werden kann (siehe Seite 48). Die im WWW verwendete Applet-Technologie bietet bereits einen Codetransfer, um Programme auf dem Client auszuführen. Mit Hilfe der Introspektion und dem JavaBeans-Standard kann der übertragene Code analysiert und generisch verwendet werden ([Orf99], [Pie99]). Zum Übertragen einzelner Objekte gibt es in Java bereits eigene Methoden zur Objektserialisierung; für die Übertragung der zugehörigen Implementierung muß allerdings ein spezieller ClassLoader entwickelt werden. Eine andere Anwendung für die Versendung von Java-Code sind die von IBM entwickelten Aglets, kleine Java-Software-Agenten, die über das Internet verschickt werden können, um auf anderen Rechnern bestimmte Aufgaben zu verrichten ([W20], [Dor97], [LaO98]).

Ein anderer Ansatz für die Umsetzung des Systems auf Java sind die sogenannten Enterprise Java Beans. Sie machen besonders intensiv Gebrauch von der Introspektion und stellen eine standardisierte Umgebung zur Verfügung, in der Klassen dynamisch erzeugt und miteinander verknüpft werden können. Besonders interessant sind hierbei die Persistenzverfahren von EJB 2.0, mit denen bestimmte Entity-Beans auf Datenbanktabellen und -klassen abgebildet werden können. Im Falle von *Container-Managed Persistence (CMP)* sorgt eine datenbankspezifische Containerklasse für die Aktivierung und Deaktivierung[30] von Instanzen, im Falle von *Bean-Managed Persistence (BMP)* ist jede Entity-Klasse selbst für die Interaktion mit der Datenbank zuständig. Für eine detaillierte Beschreibung der einzelnen Verfahren sei auf die Fachliteratur verwiesen ([Mon01], [AAG*01]).

6.2 Neuere Versionen des ODMG-Standards

Während der Arbeit an diesem Sichtsystem wurden neue Versionen des ODMG-Standards veröffentlicht. Durch teilweise grundlegende Änderungen wäre eine Anpassung der vorliegenden Arbeit zu aufwendig gewesen. Die notwendigen Änderungen sollen daher in den folgenden Unterkapiteln einzeln beschrieben werden.

6.2.1 ODMG–93, Version 1.2

Im Jahr 1996 kam Version 1.2 des ODMG-Standards heraus ([Cat96]). Die beiden wesentlichen Punkte waren dabei eine weitgehende Bereinigung des Typsystems und eine Syntaxanpassung an den SQL–92-Standard. Weiterhin sind die in den Vorversionen aufgefallenen Widersprüche zwischen den einzelnen Buchkapiteln beseitigt worden. Im folgenden sind die wichtigsten Unterschiede zur Version 1.1 aufgeführt.

Objektmodell

Das Typsystem ist vereinfacht und orthogonalisiert worden. Die Begriffe *mutable* und *immutable* wurden weitestgehend fallengelassen — der Standard verwendet jetzt, wie auch diese Arbeit, die üblichen Begriffe *Objekt* und *Literal* zur Unterscheidung. Bei der Lebenszeit eines Objekts wird nur noch zwischen *transient* und *persistent* unterschieden, die Unterscheidungen zwischen *coterminus_with_procedure*, *coterminus_with_process* und *coterminus_with_database* werden im Modell nicht mehr betrachtet; diese Details bleiben der Implementierung überlassen. Ebenfalls Aufgabe der Implementierung ist die Auflösung von Namenskonflikten bei Mehrfachvererbung. Die Klassifikation der konkreten Datentypen in Objekte und Literale ist allerdings immer noch nicht konsequent genug. Es gibt auch weiterhin strukturierte Objekte, die aber von strukturierten Literalen nicht mehr unterschieden werden können, da sie die gleichen Typnamen verwenden. Die verwendeten Typkonstruktoren (Collection, Set, List, Bag, Array) sind hierbei allesamt von Object abgeleitet, genau wie die strukturierten Literale Date, Interval, Time und Timestamp. Die einzelnen Literaltypen unterstützen nun keine Nullwerte mehr. Statt dessen gibt es für jeden Datentyp T einen zusätzlichen Datentyp nullable_T, der einen Nullwert nil hinzufügt. Ebenfalls neu ist ein spezieller Table-Datentyp, der die Arbeit mit relationalen Tabellen vereinfacht. Diese Datentypen sind aber genau wie Date, Interval, Time und Timestamp nicht in der ODL-Grammatik enthalten und können somit im Schemaentwurf nicht verwendet werden.

Anfragesprache

Die Anfragesprache ist stark an SQL–92 angepaßt worden. Um jede SQL–92-Anfrage als gültigen OQL-Ausdruck zuzulassen, wurde die Sprache um die entsprechenden Sprachelemente und

30. Passivierung im EJB-Jargon.

syntaktischen Vereinfachungen erweitert. Weiterhin wurde für jedes so erhaltene Sprachkonstrukt die funktionale Auswertungssemantik vollständig festgelegt (Variablenbindung, Sichtbarkeit, Gruppierung, Bedeutung von „select * ...“, usw.). Neben diesen SQL-spezifischen Änderungen wurden auch neue funktionale Elemente in OQL eingefügt. Hinzugekommen ist zum Beispiel ein Dereferenzierungsoperator *, der den Zustand eines Objekts als Struktur liefert. Man kann Listen mittels Intervallgrenzen erzeugen, Teile von Zeichenketten extrahieren und Zeichenketten mit Mustern vergleichen. unique() und exists() erlauben, auf einelementige und nichtleere Mengen zu testen. Außerdem können jetzt beliebige freie Funktionen aufgerufen werden, wobei Funktionen und Methoden ohne Rückgabewert (void) bei der Anfrageauswertung als nil interpretiert werden. Update-Operatoren für Attribute sind auch weiterhin nicht vorgesehen — jede Wertveränderung soll nur über benutzerdefinierte Methoden stattfinden, wobei allerdings mengenwertige Attribute durch Aufruf der auf Mengen definierten Standardmethoden bereits jetzt direkt verändert werden können.

C++-Sprachbindung

Der Begriff C++-ODL hat eine neue Bedeutung bekommen. Anstatt C++ um ODMG-spezifische Schlüsselwörter zu ergänzen, wird nun eine Standard-C++-Klassendeklaration als C++-ODL bezeichnet, wenn sie der Sprachbindung entspricht und die in ODMG definierten Datentypen verwendet. Es muß also weder ein Präprozessor verwendet noch der Compiler geändert werden, um C++-ODL zu einem lauffähigen Programm zu übersetzten. Statt dessen kann man davon ausgehen, daß eben dieses C++-ODL durch einen Präprozessor aus normalem ODL erzeugt worden ist. Die in der Sprachbindung vordefinierten Klassen sind vollständig umbenannt worden; alle erhalten jetzt einheitlich ein d_ als Präfix (d_Object, d_Set, d_Date, ...). Außerdem gibt es für jeden Literaltyp eine normierte Variante (d_Short, d_Float, d_Char, d_String, ...), die eine architekturunabhängige Darstellung des jeweiligen Typs zur Verfügung stellt und für Attribute verwendet werden soll. Die Typen können dabei wahlweise auf die vorhandenen Standardtypen (short, float, ...) abgebildet oder aber neu definiert werden. Attribute werden weiterhin durch Datenfelder ohne Lese- und Schreibmethoden dargestellt, obwohl im generischen Teil betont wird, daß Attribute nur einen „abstrakten Zustand“ darstellen, der beliebig realisiert sein kann. Für die Darstellung von Relationships werden jetzt konkrete Template-Klassen d_Rel_Ref<>, d_Rel_Set<> und d_Rel_List<> definiert, die durch Übergabe einer Zeichenkettenkonstanten befähigt werden, auf die jeweils inverse Relation zuzugreifen, um referenzielle Integrität zusichern zu können. Wesentlich ausführlicher ist auch das Fehlerverhalten spezifiziert, indem eine Reihe von C++-Exceptions definiert werden, die in den verschiedenen Fehlerfällen ausgelöst werden. Völlig neu gestaltet ist die Ausführung von OQL-Anfragen. In einem neu definierten Query-Objekt werden jetzt schrittweise die Anfrage und alle Parameter gesammelt, bevor es zur Ausführung kommt. Die Korrektheit der Parameter kann jetzt überprüft werden, die kritische Übergabe mittels Ellipse entfällt und das Funktionstemplate d_oql_execute<> ist nur noch vom Ergebnistyp abhängig. Es ist jetzt zwar aufwendiger aber wesentlich sicherer, eine Anfrage zu formulieren und auszuführen.

Auswirkungen auf das Sichtmodell

Das hier vorgestellte Sichtmodell kann prinzipiell unverändert für Version 1.2 des ODMG-Standards übernommen werden, da sich an den Grundeigenschaften des Systems nichts verändert hat. Aufwendig ist jedoch die Anpassung an die neuen Klassennamen und vor allem an die neue OQL-Syntax (zum Beispiel „select * from x as y“ statt „select struct(...) from y in x“). Für eine konforme Implementierung muß auch untersucht werden, wie sich die von mir implementierten Relationship-Klassen in die neuen Standardklassen umsetzen lassen. Ein wesentlicher Vorteil der Ver-

sion 1.2 ist, daß kompliziertere Umformungen in Funktionen ausgelagert werden können, die nun von OQL aus aufrufbar sind. Die deklarativen Herleitungen werden hierdurch mächtiger, allerdings schränken die Funktionen die möglichen Optimierungen ein, da der Inhalt der Funktionen dem Optimierer im allgemeinen nicht bekannt sein wird.

6.2.2 ODMG 2.0

Die nächste Version des Standards hieß ODMG 2.0 und wurde 1997 veröffentlicht ([CaB97]). Positiv zu beurteilen sind dabei die zusätzliche Sprachbindung für Java, die Definition eines Metaschemas und die Möglichkeit für einen serialisierten Objektaustausch (*object interchange format, OIF*). Negativ ist, wie auch in den vorangegangenen Versionen, die erneute Veränderung des Objektmodells und die wieder verstärkte Inkonsistenz zwischen den einzelnen Kapiteln. Im folgenden sind die wesentlichen Änderungen zusammengefaßt.

Objektmodell

Neben diversen Detailänderungen einzelner Datentypen sind zwei Änderungen von besonderer Bedeutung: die veränderte Interpretation von Interfaces und Klassen und das neue Metaschema. Bisher basierte die Strukturdefinition benutzerdefinierter Objekttypen auf einer Vererbungshierarchie von Interfaces; als Klasse wurde dann die Kombination eines solchen Interfaces mit einer zugehörigen Implementierung aufgefaßt. In Anlehnung an das Objektmodell von Java werden nun beide Konzepte bereits zur Spezifikation verwendet: Interfaces definieren jetzt nur noch das Verhalten, Klassen fügen die Struktur hinzu.[31] Aus diesem Grunde können auch nur Klassen instanziiert werden, nur sie haben also einen Extent. Innerhalb der Klassenhierarchie gilt nur noch Einfachvererbung: jede Klasse hat maximal eine direkte Superklasse. Für die Interfaces gilt weiterhin Mehrfachvererbung, und jede Klasse kann beliebig viele Interfaces anbieten. Die Unterscheidung zwischen Objekten und Literalen ist noch immer nicht sauber herausgearbeitet. Viele der definierten Literaltypen werden von Object abgeleitet und definieren eigene Methoden, obwohl Literale weder Objektidentität noch Verhalten aufweisen dürften. Für die Aggregattypen (Set, List, usw.) gibt es auch weiterhin nicht unterscheidbare literale und objektwertige Ausprägungen; darüber hinaus sind Dictionaries als neue Aggregattypen hinzugekommen. Neu ist außerdem die Verwendung von Objektfabriken zur Erzeugung neuer Objekte im generischen Teil des Standards (siehe 6.7.5). Die neu definierten Klassen des Metaschemas stellen zur Laufzeit Informationen über das verwendete Datenbankschema zur Verfügung. Die Definition ist detailliert und umfangreich und umfaßt alle in ODL spezifizierbaren Informationen. Die dabei verwendeten Klassennamen (Class, Interface, Exception, usw.) sind zwar eingängig, werden aber vermutlich bei ihrer Verwendung zu Problemen führen, da es sich um Schlüsselwörter von ODL und den meisten Programmiersprachen handelt.

Die Spezifikationssprache ODL wurde an das neue Objektmodell angepaßt. Es gibt nun sowohl Interface- als auch Klassendefinitionen:

```
interface anInterface : supi1, supi2
    { declarations; }

class aClass extends supc : supi1, supi2
    ( extent_spec key_spec )
    { declarations; }
```

31. Im Gegensatz zu Java dürfen Interfaces auch Attribute definieren. Diese Definitionen dienen aber nur der Zugriffsspezifikation; in der implementierenden Klasse müssen diese Attribute erneut deklariert werden.

Interfaces erben nur von anderen Interfaces, Klassen von genau einer Superklasse sowie weiteren Interfaces, deren Verhalten sie implementieren. Extent- und Schlüsseldeklarationen sind nur in Klassen möglich; die Spezifikation von persistent bzw. transient entfällt — alle von Object abgeleiteten Klassen können persistente Objekte enthalten (*persistence capable*), über die Persistenz einzelner Objekte wird bei deren Erzeugung entschieden. Neben dem neuen Dictionary-Typ sind endlich auch die strukturierten Datentypen date, time, timestamp und interval in der Grammatik zu finden; der bereits in Version 1.2 definierte Datentyp für Tabellen fehlt hingegen immer noch. Feldwertige und geordnete Relationships (array bzw. order by) entfallen ohne weitere Begründung.

Neben dieser Definitionssprache definiert der Standard nun auch ein Austauschformat für Datenbankinhalte. Das *Object Interchange Format (OIF)* basiert auf dem STEP-Standard (*Standard for the Exchange of Product Model Data*, ISO 10303) und stellt eine textuelle Repräsentation für serialisierte Datenbankobjekte dar. Eine solche mit odbdump erstellte Textdatei kann von einer anderen Datenbank mit odbload wieder eingeladen werden. Das folgende Beispiel vermittelt einen Eindruck von der verwendeten Syntax:

```
P1 Person { lname "Müller", fname "Klaus", birthdate "17.03.1927", address A1 },
P2 Person { lname "Müller", fname "Hilde", birthdate "04.08.1932", address A1 },
A1 Address { street "Schillerstraße 15", city "Hamburg", residents { P1, P2 } }
```

Anfragesprache

In der Anfragesprache OQL gibt es ebenfalls Veränderungen: benannte Anfragen werden um Parameter ergänzt und können mit delete wieder gelöscht werden, Zugriffe auf den neuen Datentyp Dictionary kommen hinzu und der in Version 1.2 eingeführte Dereferenzierungsoperator entfällt wieder; dafür wird die alte Variablen-Syntax „FROM var IN query“ als Ergänzung zu „FROM query AS var“ wieder zugelassen. Ebenfalls neu ist eine umfangreiche Spezifikation zur Konsistenz und Typkompatibilität bei der Anfrageauswertung. Ein wichtiges Detail ist, daß die strikte Auswertung boolescher Ausdrücke von links nach rechts zu Gunsten möglicher Optimierungen aufgegeben wird. Der Ausdruck „x=nil or x.age>15“ liefert jetzt undefined statt true, wenn x nil ist. Die in [CaB97] angegebene OQL-Grammatik ist allerdings identisch mit der Version 1.2, alle Änderungen sind ausschließlich im begleitenden Text zu finden!

C++-Sprachbindung

In der C++-Sprachbindung sind die folgenden Detailänderungen vorgenommen worden: Verwendung von d_Boolean statt int, Implementierung von d_Dictionary, Unterstützung mehrfacher Objektnamen, Thread-Unterstützung bei Transaktionen, Unterstützung von Datentypen der *Standard Template Library (STL)*, obligatorische Verwendung von d_Object, automatische Extent-Verwaltung, Wiedereinführung von d_Collection::query, spezielle Iterator-basierte Anfragemethode für große Ergebnismengen und Definition eines Metaschemas. Auffällig ist, daß sich das Objektmodell der Sprachbindung jetzt stärker vom generischen unterscheidet. Weder die Objektfabriken noch die Strukturierung in Interfaces und Klassen haben Eingang in die Sprachbindung gefunden; im Gegenteil wird der Sprachunterschied von C++ zu ODL besonders betont, indem das im generischen Teil definierte Metaschema an die Eigenschaften von C++ angepaßt wird: Interfaces werden nicht modelliert, Methoden müssen nicht an Klassen gebunden sein, Attribute und Methoden können statisch und geschützt sein und die Vererbungshierarchie berücksichtigt virtuelle und geschützte Superklassen. Hier wird wieder der Zwiespalt von ODMG deutlich, einerseits einen generischen, systemunabhängigen Standard zu definieren und andererseits die Sprachbindung hundertprozentig in die verwendete Sprache einzubetten.

Java-Sprachbindung

Die Sprachbindung für die Programmiersprache Java ist in ODMG–2.0 erstmals definiert. Wie bei C++ handelt es sich um eine einbettende Abbildung, wobei verschiedene Möglichkeiten zur Persistenzbereitstellung diskutiert werden (Präprozessor, Bytecode-Konverter oder neue virtuelle Maschine). Die Abbildung ist bisher noch unvollständig und erlaubt viele Freiheiten. Relationships, Extents, Schlüssel, Aufzählungen, Intervalle und Dictionaries werden nicht unterstützt, Literale können wahlweise auf primitive Datentypen oder die korrespondierenden Objekttypen abgebildet werden. Auf die Abbildung von Interfaces wird nicht eingegangen, ein Metaschema ist noch nicht verfügbar. Für eine konkretere Definition wird man auf einen neuen Standard warten müssen (siehe Abschnitt 6.2.3).

Auswirkungen auf das Sichtmodell

An den grundlegenden Eigenschaften des Objektmodells hat sich nichts geändert. Sowohl im generischen Teil als auch in der Sprachbindung bleibt die unveränderbare Ein-Klassen-Zugehörigkeit erhalten, so daß auch weiterhin mit neu zu erzeugenden Sichtobjekten gearbeitet werden muß. Für Interfaces ist keine besondere Sichtunterstützung notwendig, da sie nur Schnittstellen aber keine Implementierungen definieren. Soll eine Sichtklasse ein Interface implementieren, so muß die Herleitung der einzelnen Methoden in der Klasse selbst erfolgen, genau wie jede Interface-Methode letztendlich in der Klasse implementiert werden muß. Dies ist insofern sinnvoll, als die Herleitungen überlicherweise auf einer Navigation über das orig-Attribut beruhen, das erst in der Klasse selbst definiert wird. Da ODMG–2.0 allerdings die redundante Attributdeklaration in Interfaces zuläßt, kann man auch die Herleitungsdefinition in Interfaces in Betracht ziehen, muß jedoch mit einem erhöhten Konfliktpotential rechnen. Neben den Interfaces muß im Sichtentwurf auch der Wegfall von Mehrfachvererbung beachtet werden; angesichts der allgemeinen Probleme bei Mehrfachvererbung in C++ (virtuelle Superklassen, Namenskonflikte, variable Objektadressen, usw.) ist ein solcher Verzicht aber sowieso anzuraten. Bei der Spezifikation muß man virtuelle Klassen jetzt als class und nicht mehr als interface deklarieren. Mit dem Wegfall der Persistenz-Schlüsselwörter persistent und transient kann das Schlüsselwort virtual zu Gunsten der Lesbarkeit auch vor die Klassendeklaration verschoben werden:

```
virtual class viewclass { ... };
```

Bei der Formulierung der Herleitungsklauseln muß man die neue Auswertungssemantik für Nullwerte beachten und kann gegebenenfalls parametrisierte benannte Anfragen für ähnliche Herleitungen ausnutzen. Bei der Sprachbindung sind die Detailveränderungen der einzelnen Datentypen zu beachten, außerdem kann man jetzt statt des selbst definierten Data Dictionary das beigefügte Metaschema verwenden. Zur Ablage der Herleitungsklauseln muß man entweder das Metaschema um ein entsprechende Datenfelder erweitern, oder man verwendet einfach das bereits vorgesehene Kommentarfeld zur Ablage dieser Information.

6.2.3 ODMG 3.0

Die neueste Version, 3.0, ist Anfang 2000 erschienen und heißt jetzt nur mehr „Object Data Standard“ ([CaB00]). Er wird von der ODMG als stabil angesehen und soll bis auf weiteres nicht mehr geändert werden. Die Mehrzahl der Änderungen im Vergleich zur Version 2.0 betrifft nur Details, wie etwa neue Methoden oder geänderte Parameterlisten (zum Beispiel ein neuer long long Typ oder sogenannte *Visitors* zum Iterieren über das Metaschema). Die Grammatiken sind teilweise erneuert und verbessert worden und einige Inkonsistenzen wurden beseitigt. Die folgende Liste enthält alle größeren Änderungen:

- Der any-Datentyp entfällt ohne weiteren Hinweis und wird durchweg durch Object ersetzt. Wie dies im Zusammenhang mit Literalen funktionieren soll wird nicht erklärt.
- Nullwerte werden ebenfalls nicht mehr unterstützt. Die in Version 2.0 eingeführten „nullable types“ entfallen. Statt dessen wird ausführlich spezifiziert, wann Anfrageergebnisse als undefiniert anzusehen sind, und wie man mit ihnen weiter umgeht.
- In OQL kommen zwei neue Operatoren hinzu, die eine geordnete Auswertung boolescher Ausdrücke erzwingen: andthen und orelse. Hierdurch wird die in Version 2.0 entfallene Shortcut-Auswertung wieder ermöglicht.
- Die Java-Sprachbindung ist vervollständigt worden. Die wichtigste Neuerung ist dabei die Verwendung einer sogenannten Profildatei, in der außerhalb des Java-Codes festgelegt wird, welche Klassen persistente Objekte enthalten können und welche Attribute zu einer Relationship gehören. Die virtuelle Maschine ist dann dafür zuständig, diese Datei zu lesen, zu interpretieren, und die notwendige Funktionalität zur Verfügung zu stellen. Speziell bedeutet dies, daß die referenzielle Integrität nicht von einer Relationship-Implementierung realisiert wird, sondern daß das System selbst erkennen muß, wann ein objektwertiges Attribut geändert worden ist, und welche Maßnahmen zur Integritätserhaltung notwendig sind. Eine solche Profildatei sieht wie folgt aus:

```
; Globale Eigenschaft: alle Klassen sollen persistente Objekte enthalten dürfen
class *
persistent=capable
; Spezifische Eigenschaften der Klasse author:
class my.package.author
field lastname
transient=false
field publications
refersTo=Publication
inverse=authors
...
```

An der C++-Sprachbindung gibt es keine Änderungen. Das bedeutet, daß zum Beispiel der neue long long Typ nicht unterstützt wird, und daß die Abbildung von ODL auf C++ noch immer nicht zufriedenstellend definiert ist.

Das neue Java-Package „org.odmg“ ist über [W11] erhältlich. Bereits im Februar 1999 wurde unter [W9] berichtet, daß beim ODMG-Treffen im November 1998 beschlossen wurde, neben einer objektrelationalen Sprachbindung auch eine solche für XML zu entwickeln. Dies bezieht sich vermutlich eher auf XML als Austauschformat. Eine entsprechende Spezifikation von Mai 2000 ist unter [W12] verfügbar.

Auswirkungen auf das Sichtmodell

Da es nur geringfügige Änderungen im Objektmodell und überhaupt keine Änderungen an der Sprachbindung gibt, hat die Version 3.0 im Vergleich zur Version 2.0 keine weiteren Auswirkungen auf das Sichtmodell.

6.3 Relationale Datenbanken

Die Anwendung des vorgestellten Sichtmodells auf relationale Datenbanken erscheint nicht sinnvoll. Relationale Systeme bieten bereits seit langem standardisierte und ausgefeilte Sichtmechanismen, die das Datenmodell angemessen unterstützen und effizient ausgewertet werden

können. Es gibt keine Objektidentität, die erhalten oder abgebildet werden müßte, und es gibt keine Klassenstrukturen und Methoden. Ein mögliches Einsatzgebiet ergibt sich jedoch, wenn relationale Datenbanken mit einer objektorientierten Schnittstelle versehen werden. Im Rahmen von IRO–DB ist zum Beispiel ein ODMG-Adapter für das relationale Datenbanksystem Ingres entworfen worden, das unter Berücksichtigung von Schlüsseln und Fremdschlüsseln für eine Menge relationaler Tabellen ein ODMG-Schema mit Klassen und Relationships erzeugt und die entsprechende Datenabbildung vornimmt (ibermática [W13], siehe auch [FaV95]). Es existiert bereits eine Vielzahl an Systemen, die relationale Tabellen in objektorientierter Form zur Verfügung stellen (zum Beispiel Subtleware (Subtle Software), CocoBase (Thought), JavaBlend (Sun Microsystems) oder Relational Object Framework (Watershed Technologies)) bzw. ein Objektmodell in einer relationalen Datenbank persistent ablegen (zum Beispiel Persistence (Persistent Data Systems) oder Tangram (University of Winnipeg)). Eine umfassende Sammlung solcher Systeme ist auf der Webseite von Cetus-Links [W14] zu finden; außerdem arbeitet die ODMG selbst an einer relationalen Abbildung ihres Modells [W9]. [RBP*91] zeigt als Anwendungsbeispiel ebenfalls eine mögliche Abbildung des OMT-Objektmodells auf relationale Datenbanken. Durch diese Abbildungen erhält man letztendlich persistente Objektmodelle, auf die das vorgestellte System wiederum angewandt werden kann (siehe unten).

6.4 Objektrelationale Datenbanken — SQL3

Für objektrelationale Datenbanken gilt sinngemäß das gleiche wie für relationale Datenbanken. Bei den derzeit verfügbaren Systemen, wie etwa Informix Dynamic Server (Informix), DB2 (IBM) oder Oracle 8*i*/9*i* (Oracle Corporation), handelt es sich um relationale Systeme mit derzeit noch geringen objektorientierten Erweiterungen. Die Objektorientierung beschränkt sich im allgemeinen auf die Fähigkeit, abstrakte Datentypen als Objekte zu definieren und in Tabellen abzulegen. Die Interaktion mit der Datenbank erfolgt weiterhin auf Basis von SQL. Die eingebetteten Objekte werden dabei entweder unverändert ins Ergebnis übernommen, um direkt von der Anwendung verwendet zu werden, oder sie dienen der Umsetzung geschachtelter Tabellen, die zum Beispiel mit den Konstrukten aus SQL3 abgefragt werden können. Durch diese relationale Ausrichtung ist die Anwendung meines Sichtsystems auch für objektrelationale Datenbanken nicht sinnvoll. Die oben geführte Diskussion zur Verwendung relationaler Datenbanken als Basis für persistente Objektsysteme gilt natürlich auch für objektrelationale Datenbanken.

6.5 Objektorientierte Datenbanken und Objektsysteme

Das hier vorgestellte Sichtsystem basiert speziell auf dem Datenmodell von ODMG und dessen Umsetzung auf C++. Die Grundidee der virtuellen Klassen läßt sich hierbei aber auch auf andere objektorientierte Datenmodelle anwenden, wie zum Beispiel objektorientierte Datenbanken wie O_2 (Ardent Software), POET (Poet Software), Versant (Versant Object Technology) oder ObjectStore (Object Design), oder die oben erwähnten persistenten Objektsysteme auf Basis relationaler oder objektrelationaler Datenbanken. Eine Anpassung der konkreten Lösung mit Spezifikation, Präprozessor und Laufzeitumgebung an das gewählte System kann dabei allerdings sehr aufwendig werden. Zu beachten ist, daß das System die Existenz einer Anfragesprache voraussetzt. Verwendet man im Laufzeitssytem ausschließlich die prozedurale Darstellung der Sichtobjekte (siehe Abschnitt 3.5), kann man diese Voraussetzung umgehen, sofern man für die in der prozeduralen Darstellung noch auftretenden Anfragen (zum Beispiel zur Extentbestimmung) geeignete Methoden zur Verfügung stellt. In den folgenden beiden Abschnitten werden zwei Objektsysteme, nämlich VODAK und CORBA, näher untersucht.

6.6 VODAK

VODAK ist ein objektorientierter Datenbankprototyp, der am Institut für Integrierte Publikations- und Informationssysteme (IPSI) [W1] entwickelt worden ist ([Kla93], [KFM*96], [W4]). Beim Entwurf wurde großer Wert darauf gelegt, das System offen und erweiterbar zu definieren. Im Gegensatz zur C++-Sprachbindung der ODMG verwendet VODAK eine indirekte Sprachbindung und ein eigenständiges OID-Konzept. In den Modellierungs- und Anfragesprachen VML und VQL kann man wie gewohnt mit Objekten arbeiten. Durch einen Präprozessor wird dieser Code dann in C++ übersetzt, wobei Objektzugriffe durch sogenannte *Messages* ersetzt werden, die an die jeweilige OID geschickt werden. Ein speziell entworfener Message-Handler übernimmt dabei die Abbildung solcher Messages auf den zugehörigen C++-Code. Durch die Definition von *Metaklassen* kann dabei auf den Message-Handler Einfluß genommen werden, indem zum Beispiel Vererbungshierarchien wie Spezialisierung oder Generalisierung definiert werden ([KNS90]).

Durch die Flexibilität der indirekten Abbildung ist es prinzipiell möglich, in VODAK ein objekterhaltendes Sichtsystem zu implementieren. Eine OID wäre dabei mehreren Klassen (Rollen) zugeordnet, und der Message-Handler müßte hieraus die jeweils aktuelle Klasse auswählen. Die korrekte Auswahl einer Klasse und die notwendigen Kontextwechsel für die verschiedenen Rollen eines Objekts können dabei ziemlich kompliziert werden (vgl. Up/Down-Funktionen in [HeZ88b]). Zur Unterstützung könnte dabei ein Teil der OID zur Codierung von Kontext-Information verwendet werden: Alle Rollen eines Objekts verwenden dabei die gleiche Grund-OID, unterscheiden sich aber im Kontext-Code. Jede Objektreferenz kennt hierdurch ihre aktuelle Rollenzuordnung, und Kontextwechsel können durch Manipulation der OID erreicht werden. Beim Vergleich von OIDs muß in diesem Fall der Kontext-Code ausgeblendet werden, um Objektidentität zwischen den Rollen zu behalten.

Mit Hilfe der Metaklassen kann aber auch der vorgestellte verallgemeinerte identitätserhaltende Sichtansatz in VODAK zur Verfügung gestellt werden. Die Herleitung einer Sichtklasse aus Daten einer bereits existierenden Klasse kann durch eine VIEW_METACLASS geschehen, die die Beziehung zwischen diesen Klassen definiert. Für die Implementierung einer solchen Metaklasse muß untersucht werden, wie man die konkrete Herleitung einer Sichtklasse aus deren Originalklassen modelliert. Dies kann innerhalb der Sichtklassenimplementierung, durch Parametrisierung der Metaklasse oder durch die Implementierung verschiedener Metaklassen erfolgen.

Eine weitergehende Untersuchung dieser Möglichkeiten ist nicht geplant, da die Weiterentwicklung von VODAK eingestellt wurde.

6.7 Übertragung auf CORBA

Eine Übertragung des Sichtsystems auf die verteilte Objektumgebung CORBA ist bereits ansatzweise durchgeführt worden ([RBPF98], [RBPF99]). Nach einer kurzen Einführung werden Ansatz, Architektur und Erweiterungsmöglichkeiten der Sichtübertragung beschrieben.

6.7.1 CORBA

Die *Object Management Group OMG* ([W5]) ist ein Konsortium von derzeit mehr als 800 Mitgliedern, das mit dem Ziel gegründet wurde, einen offenen Standard für objektorientierte Systementwicklung zu definieren, mit dem plattformunabhängiger Zugriff auf verteilte Objekte

möglich ist. Die Objektschnittstellen werden in der Spezifikationssprache *IDL* entworfen, aus der dann Implementierung und Zugriffsverfahren generiert werden. Der verteilte Zugriff auf die Objekte erfolgt dabei über eine Art „Software-Bus“, den *Object Request Broker* (*ORB*, Bild 23), dessen Detailarchitektur (*Common Object Request Broker Architecture, CORBA*) derzeit in Version 2.6 vorliegt[32] ([OMG91], [OMG99]).

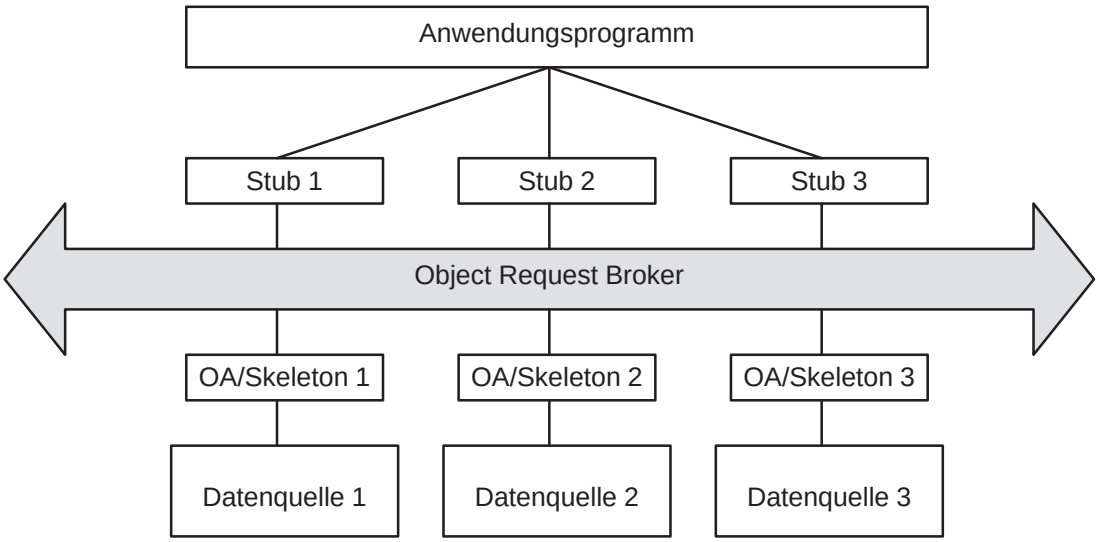


Bild 23: Verteilter Datenzugriff mit CORBA

Die Gesamtarchitektur des System ist in der *Object Management Architecture OMA* ([OMG95a]) festgelegt und definiert zusätzliche Schichten über dem ORB (Bild 24). Vordefiniert sind die sogenannten *CORBAservices (COSS,* [OMG97]), die unterschiedliche Gundfunktionen zum System hinzufügen, wie etwa Transaktionen, Verzeichnisdienste, Anfragebearbeitung und Benachrichtigungsmodelle. Ausgehend von den Services können komplexere Module bereitgestellt werden, die Anwendungsprogrammen umfassendere Funktionen anbieten. Die *Horizontal Facilities* sind dabei unabhängig vom Anwendungsgebiet — hierzu zählen zum Beispiel Benutzungsoberflächen und Dokumentenverwaltung. Die *Vertical Facilities* stellen jeweils Komplettlösungen für bestimmte Anwendungsgebiete, wie etwa Versicherungs- oder Gesundheitswesen, zur Verfügung ([OMG95b]).

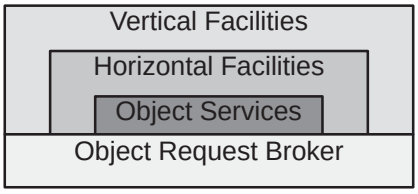


Bild 24: Object Management Architecture (OMA)

Ein anderes System für verteilte Objektzugriffe ist das *Distributed Component Object Model (DCOM)* von Microsoft, das als Weiterentwicklung von OLE/COM die Kommunikation zwischen ActiveX-Komponenten steuert ([Ses97], [W15]). Im Gegensatz zum offenen CORBA-Standard, der von verschiedenen Herstellern auf unterschiedlichen Plattformen realisiert werden kann, handelt es sich hierbei um eine herstellerspezifische Implementierung, deren Schnittstellen nicht offengelegt sind. Die proprietäre Software ist nur auf der Windows-Plattform lauffähig

32. CORBA 3.0 wird bereits seit mehr als drei Jahren angekündigt, ist jedoch immer noch nicht erschienen ([W6]).

und nur im Rahmen der EntireX-Produktreihe [W16] auf Unix portiert. Einen Vergleich von COM und CORBA findet man zum Beispiel in [OHE96] und [W19].

Ein anderer offener Standard für verteilte Anwendungen ist das *Distributed Computing Environment (DCE)* ([W18]) der Open Software Foundation (OSF) ([W17]). Im Gegensatz zu CORBA handelt es sich aber nicht um einen Objektstandard, sondern um eine Umgebung für verteilte Funktionsaufrufe (remote procedure call, RPC) mit zusätzlichen standardisierten Zeit- und Namensdiensten ([Loc94]). Zu beiden Standards werden in [OMG99] Schnittstellen definiert.

6.7.2 Sichtsystem als integrierende Zwischenschicht

Obwohl die Spezifikation von ODMG–93 auf CORBA basiert ([Cat94, S. 50]), ist das mögliche Zusammenwirken von ODMG–93 und CORBA noch nicht endgültig entschieden. Auf Grund des erweiterten Typsystems und der datenbankspezifischen Funktionen von ODMG–93 ist eine direkte Verknüpfung der beiden Systeme nicht so leicht möglich. Als mögliche Integration wird die Verwendung der ODMG-Elemente in Form von spezialisierten Objektadaptern untersucht ([Cat94, Appendix B], [OHE96]).

Die in dieser Arbeit vorgestellte Grundidee, virtuelle Objekte als Sichten auf reale Objekte zu verwenden, kann aber auch direkt auf CORBA angewandt werden. CORBA stellt zwar bereits eine verteilte Objektumgebung zur Verfügung, jedes Anwendungsprogramm muß aber explizit auf jede einzelne Datenquelle zugreifen. Ein Sichtsystem, das diese direkte Quellenabhängigkeit versteckt und Objekte anpassen und integrieren kann, ist daher auch hier von Vorteil. Bild 23 zeigt die in CORBA übliche Architektur: Für jede Datenquelle, auf die über den ORB zugegriffen werden soll, muß aus der Schnittstellendefinition ein sogenannter *Stub* generiert werden, der zur Anwendung hinzugebunden werden muß und die Kommunikation via ORB mit der entsprechenden Datenquelle übernimmt.[33] Durch Sichten als zusätzliche Integrationsschicht können die verschiedenen Stubs zu einem einzigen zusammengefaßt und bereits an die Anforderungen der Anwendung angepaßt werden (Bild 25).

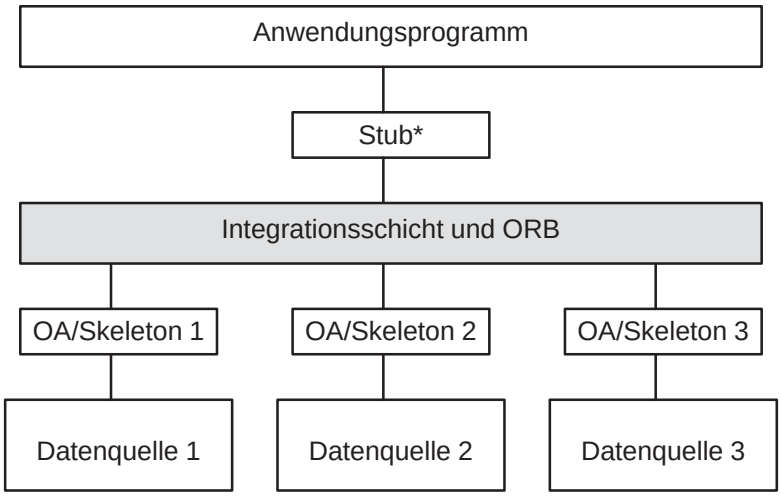


Bild 25: Integrierter Datenzugriff mit CORBA (schematisch)

33. Für die Anbindung der Datenquelle an den ORB wird zusätzlich ein *Skeleton* generiert, das zusammen mit einem generischen Objekt-Adapter (OA) den Datenaustausch ermöglicht. Durch das *Dynamic Invocation Interface (DII)* kann man auch ohne Stubs arbeiten, muß aber für jeden Objektzugriff umfangreiche Strukturen aufbauen.

6.7.3 Architektur

Genau wie im Datenbankfall wird das Sichtsystem als neue Schicht zwischen der Anwendung und den Datenquellen eingefügt. Während im IRO–DB-System eine eigene Netzwerkkomponente für den Datenaustausch entwickelt werden mußte, stellt CORBA bereits den ORB zur Verfügung. Dieser kann auch für den Zugriff der Anwendungsprogramme auf das Sichtsystem verwendet werden. Im Gegensatz zum Datenbankfall wird die Laufzeitumgebung also nicht zur Anwendung hinzugefügt, sondern steht in Form von CORBA-Dienstobjekten zur Verfügung, auf die wie üblich über Stubs zugegriffen wird. Sowohl die Designumgebung als auch die Laufzeitumgebung werden als eigenständige Module in die CORBA-Architektur eingefügt. Gemäß der Terminologie handelt es sich bei beiden um Horizontal Facilities. Die Gesamtarchitektur ist in Bild 26 angegeben.

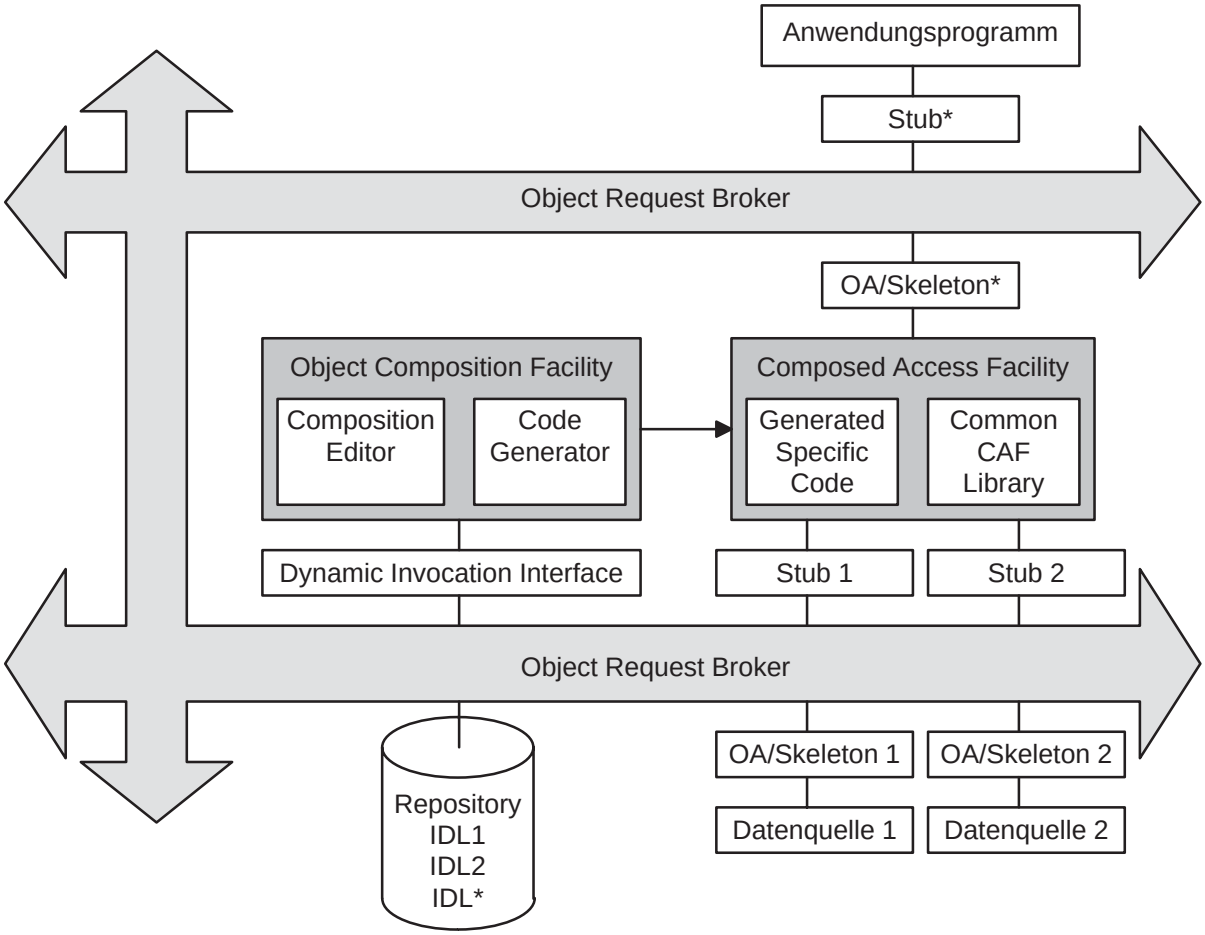


Bild 26: Architektur zur Datenintegration mit CORBA

Die *Object Composition Facility* entspricht der Designumgebung aus Abschnitt 4.6.1. Sie besteht wie bisher aus einem graphischen Entwurfswerkzeug und einem Code-Generator, die jedoch an das Datenmodell und die Sprachbindung von CORBA angepaßt werden müssen. Das *Dynamic Invocation Interface* von CORBA kann dabei verwendet werden, um aus dem *Repository,* der Schnittstellen-Datenbank des ORB, die derzeit registrierten Datenquellen zu bestimmen, und nach erfolgter Integration dort die integrierte Datenquelle zu registrieren.

Die *Composed Access Facility* stellt das eigentliche Sichtsystem, also die Laufzeitumgebung zur Verfügung. Genau wie bisher kombiniert sie schemaspezifisch generierten Code mit gemeinsa-

men Funktionsbibliotheken, die in der *Common CAF Library* zusammengefaßt sind. Sie umfassen wieder Objektverwaltung, Metainformation, Transaktionsdienste und Anfragebearbeitung, wobei auf das Zusammenspiel mit den in CORBA definierten Standarddiensten zu achten ist (Repository, QueryService, TransactionService, ConcurrencyService, Security).

6.7.4 Klassenhierarchie

Das Konzept der virtuellen Objekte wird unverändert übernommen: Aus den Schnittstellendefinitionen der Datenquellen werden Sichtklassen erzeugt, die mit Sichtobjekten instanziiert werden und über Rückwärtsreferenzen den Bezug zu den Originalklassen erhalten. Da CORBA über die in den Stubs definierten Proxy-Klassen einen transparenten Zugriff auf externe Objekte ermöglicht, können die Importklassen entfallen. Statt dessen besteht aber die Möglichkeit, eine zusätzliche Schicht aus Cache-Klassen einzufügen, um ständige ORB-Zugriffe durch die Zwischenspeicherung von Attributwerten zu vermeiden ([Emp97]).[34] Die Proxy- und Cache-Klassen werden dann mit Hilfe von Konvertierungsklassen an das gewünschte Zielschema angepaßt. Die IDL-Spezifikation der so erhaltenen Sichtklassen wird im ORB registriert und an die Anwendung weitergereicht, die dann daraus Stubs generieren und somit via ORB auf das Sichtsystem zugreifen kann. Wie im Datenbankfall sind die Zugriffsmethoden der Sichtklassen so implementiert, daß sie mit Hilfe der Rückwärtsreferenzen die entsprechenden Operationen auf den Basisobjekten ausführen.

6.7.5 Startobjekte

Ein wesentlicher Unterschied zwischen ODMG–93 und CORBA ist die Granularität, mit der mit Objekten umgegangen wird. ODMG arbeitet vorwiegend mengenorientiert: Mit Hilfe von Anfragen kann man komplexe Objektmengen bestimmen und bearbeiten, und über den Extent kommt man an alle Instanzen einer Klasse. CORBA hingegen ist eher auf die Arbeit mit einzelnen Objekten ausgerichtet. Die in IDL spezifizierten Schnittstellen definieren bestimmte Funktionalitäten, die über ein Objekt, das diese Schnittstelle anbietet, angesprochen werden können. Die hierzu notwendige Verfolgung der Objektreferenz, die Suche nach der Implementierung auf der externen Datenquelle, das Ausführen der externen Operation und die Rückübertragung der Ergebnisse an den Aufrufer werden von CORBA automatisch ausgeführt. Sobald man eine Objektreferenz hat, kann man Funktionen auf ihr ausführen und zu anderen Objekten navigieren. Problematisch ist hingegen die Bestimmung eines Startobjekts, über das man Zugriff auf die Datenquelle erhält. In CORBA gibt es kein Anfragesystem zur deklarativen Suche nach Objekten. Statt dessen muß jede Datenquelle ihre Objekte explizit in einem Namensdienst registrieren; kennt eine Anwendung den Namen eines Objekts, so erhält sie vom Namensdienst die zugehörige Referenz und kann dann mit dem Objekt arbeiten. Registriert eine Datenquelle beim Namensdienst alle Objekte, die sie dem System zur Verfügung stellen will, so kann eine Anwendung direkt auf jedes gewünschte Objekt zugreifen, dessen Namen sie kennt. Wenn eine Datenquelle jedoch sehr viele Objekte zur Verfügung stellt, ist eine solche erschöpfende Registrierung zu aufwendig. In diesem Fall bietet sich statt dessen die Verwendung von Objektfabriken an. Eine *Objektfabrik* (*object factory*) in CORBA ist ein ausgezeichnetes Objekt, mit dessen Hilfe andere Objekte oder Referenzen zu bestehenden Objekten erzeugt werden können. Definiert eine Datenquelle eine Objektfabrik, mit deren Hilfe alle anderen Objekte der Datenquelle erreicht werden können, so kann durch deren Registrierung allen Anwendungen der Zugriff auf alle Objekte

34. Manche ORBs bieten selbst Cache-Funktionalität an, wie etwa die *smart proxies* von IONA's orbix, sie verwenden aber proprietäre Schnittstellen, da CORBA kein Caching vorsieht.

ermöglicht werden. Ist für jede Datenquelle eine Objektfabrik bekannt, kann auch für das Sichtsystem eine Objektfabrik zur Verfügung gestellt werden, die die anderen Fabriken integriert und somit den Zugriff auf alle Objekte in der Sicht ermöglicht. Speziell für die Arbeit mit vielen Objekten sieht CORBA besondere Objekt-Adapter vor (*library object adapter (LOA), object database adapter (ODA)*), die zum Beispiel durch Gruppierung von Objekten effizienter arbeiten als normale Objekt-Adapter ([OHE96]).

Bei diesem Sichtansatz gehe ich davon aus, daß die zu integrierenden Objekte feingranular (fine grained) von den Datenquellen zur Verfügung gestellt werden. Das heißt, daß die verschiedenen Funktionen und Datensätze in Form einzelner Objekte definiert sind, auf die direkt zugegriffen werden kann. Der Zugriff auf die Datenquellen ist folglich schemaabhängig, da die einzelnen Objekte mit ihren jeweiligen Schnittstellen direkt exportiert werden. Weiterhin muß man mit vielen kleinen Objekten umgehen. Ein anderer Ansatz ist die grobgranulare (coarse grained) Integration, bei der man davon ausgeht, daß die Datenquellen nur wenige Objekte exportieren, diese dafür aber mächtige Manipulationsfunktionen anbieten. Ein feingranularer Export einer relationalen Datenbank definiert zum Beispiel für jede Tabelle eine Klasse und stellt jeden einzelnen Datensatz als Objekt zur Verfügung. Ein grobgranularer Export hingegen stellt für jede Tabelle nur ein Objekt zur Verfügung, das Anfragen zum Inhalt dieser Tabelle auswertet und in Form von Literalen zurückliefert. Noch gröber ist der Export eines einzigen Datenbankobjekts, an das man beliebige SQL-Anfragen schicken kann. Ein Beispiel für ein grobgranulares Integrationssystem auf Basis von CORBA ist das MIND-System von der Universität Ankara ([KOD*95], [DDK*96]). MIND ist dabei speziell für die Integration von relationalen und objektorientierten Datenbanken entwickelt worden. Jede dieser Datenbanken exportiert nur ein einziges Datenbankobjekt, an das Anfragen geschickt werden können. Die eigentliche Integration geschieht dann traditionell durch Anfrageexpansion und -verteilung. Für dieses Anwendungsgebiet stellt sich der grobgranulare Ansatz als sehr vorteilhaft heraus, da man keinerlei Objektverwaltungsaufwand hat, die Daten blockweise transportieren kann und eine deklarative Anfragesprache für den Zugriff auf die Datenquelle zur Verfügung hat ([DDK*96]). Nachteilig ist jedoch, daß nur solche Datenquellen integriert werden können, die ein Anfragesystem besitzen und dieses im erwarteten Format exportieren. Der feingranulare Ansatz hingegen erlaubt die Integration beliebiger Informationsquellen, sofern sie mittels IDL exportierbar sind. Neben Datenbanken umfaßt dies zum Beispiel auch Prozeß-Systeme (OLAP), Dateisysteme oder Funktionsbibliotheken. Außerdem können feingranulare Objekte besser in Anwendungen integriert werden, und bei der Sichtimplementierung hat man die volle Mächtigkeit einer Programmiersprache zur Verfügung, während der in MIND verfolgte Ansatz durch die Mächtigkeit der Anfragesprache eingeschränkt ist.

6.7.6 Anfragesystem

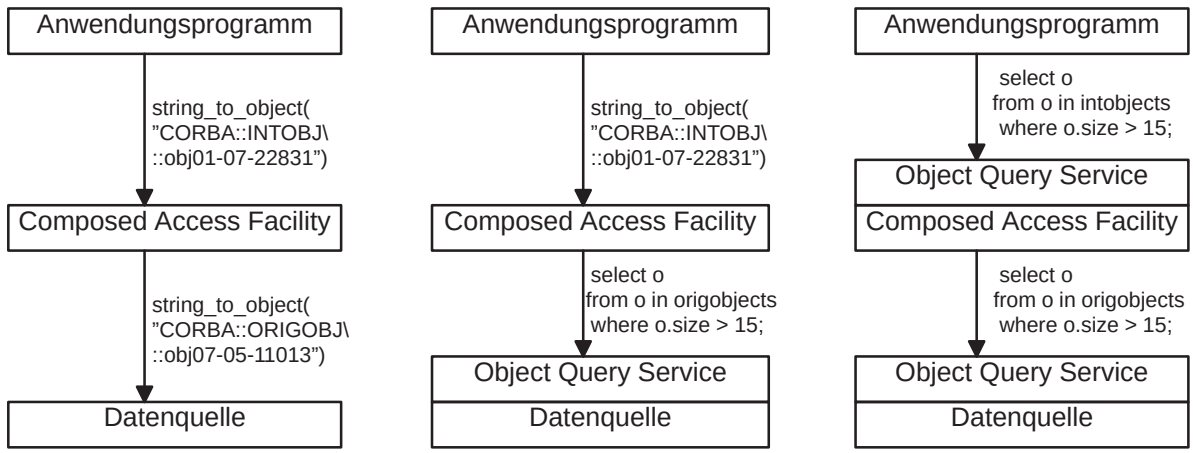
Das CORBA-Modell basiert auf der direkten Manipulation einzelner Objekte. Im Gegensatz zum mengenorientierten Datenbankmodell der ODMG stellt CORBA daher in seiner Kernfunktionalität keine Unterstützung für Objektmengen und kein Anfragesystem zur Verfügung. Entsprechende Dienste zur Bereitstellung solcher Funktionen sind im Standard aber bereits vorgesehen.

Der *Object Collection Service* ([IBM96]) ist von IBM als Vorschlag an die OMG eingereicht worden und im Juni 2000 als Version 1.0 veröffentlicht worden. Dieser Dienst definiert eine umfassende Sammlung von Datentypen zur Objektmengenverwaltung. Ausgehend von einer Basisklasse Collection wird eine vollständige Hierarchie aus geordneten und ungeordneten Mengen, Multimengen (bag) und Zuordnungen (map) sowie Stapelstrukturen (stack, queue) definiert. Je-

de dieser Strukturen verwaltet Objektmengen von genau einem Typ und definiert Standardmethoden zum Einfügen, Auslesen, Ersetzen und Löschen von Elementen. Für die schrittweise Abarbeitung werden verschiedene Iteratoren definiert. Die Erzeugung neuer Strukturen erfolgt über eine parallele Hierarchie von *CollectionFactories*. Die Funktionalität dieser Spezifikation ist direkt mit den ODMG-Mengenstrukturen, oder den entsprechenden Definitionen der *C++ Standard Template Library (STL)* ([MuS96]) oder des *Rogue Wave Tools.h++ Toolkit* ([Sun95]) zu vergleichen.

Wichtiger für ein Anfragesystem ist hingegen der *Query Service*, der bereits länger in der COS-Spezifikation ([OMG97]) enthalten ist. Mit dem Query Service ist es möglich, deklarative Anfragen wie bei einem Datenbanksystem auszuführen. Die dabei verwendete Anfragesprache und der tatsächliche Ablauf der Ausführung bleiben dabei der Implementierung überlassen. Die Spezifikation definiert nur die Schnittstelle zum Aufruf einer solchen Auswertung. Jede konforme Implementierung muß entweder SQL-92 oder OQL auswerten können; darüber hinaus können beliebige andere Anfragesprachen unterstützt werden. Die Auswertung basiert auf einem QueryEvaluator-Objekt, dem man den Anfragetext in einer der unterstützen Anfragesprachen übergibt. Der QueryEvaluator ist dann für die vollständige Auswertung der Anfrage und die Rückgabe der Ergebnisse an den Aufrufer verantwortlich; hierbei können Teilanfragen an untergeordnete QueryEvaluator-Objekte delegiert werden. Eine QueryableCollection ist ein Spezialfall eines QueryEvaluators, der Anfragen auf einer bestimmten Objektmenge vornimmt und die Auswertung gegebenenfalls automatisch an alle Elemente delegieren kann. Für komplexere Anfragesituationen wird ein QueryManager verwendet, der Teilauswertungen einzelner Query-Objekte miteinander koordiniert. Durch die Implementierungsfreiheit und die ausdrücklich vorgesehene Möglichkeit zur Anfrage-Delegation können so komplexe Auswertungssysteme implementiert werden, die für unterschiedliche Umgebungen optimiert werden können. Eine ebenfalls in der Spezifikation enthaltene Klassenbibliothek für Objektmengen soll bei Verfügbarkeit des Object Collection Service durch dessen Definitionen ersetzt werden ([OMG97], [OHE96]).

Bisher sind leider noch keine Implementierungen von Query Services verfügbar; bei einer Umsetzung des Sichtsystems auf CORBA kann daher derzeit noch nicht auf Anfragefunktionalität zurückgegriffen werden. Es bietet sich daher eine Phasung an, bei der Anfragefunktionalität schrittweise hinzugefügt wird (Bild 27). Die erste Implementierung basiert auf der expliziten Integration einzelner Objekte. Ausgehend von einem Startobjekt, das man zum Beispiel über einen Namensdienst ermittelt, kann man direkt zu anderen Objekten navigieren und diese verändern oder miteinander kombinieren. Die so erzeugten Objekte werden wiederum beim Namensdienst registriert und stehen somit einer Anwendung zur Verfügung. Beim Zugriff auf ein solches Objekt wird dann direkt zu den Originalobjekten navigiert (27a). Sobald ein Query Service für eine Datenquelle zur Verfügung steht, kann man den navigatorischen Zugriff auf Originalobjekte durch deklarative Anfragen ersetzen. Der Zugriff auf die Sichtobjekte erfolgt weiterhin direkt, die Definition der Sichtobjekte und deren Zugriff auf Originalobjekte können aber von der Anfragefunktionalität der Datenquellen profitieren (27b). In einem letzten Schritt kann dann ein Query Service hinzugefügt werden, der es erlaubt, auch Sichtobjekte über einen deklarativen Zugriff zu erhalten. Die Anfragen werden hierbei an die CAF gerichtet, die die Sichtdefinitionen auswertet und deren Herleitungsklauseln an die ursprünglichen Datenquellen delegieren kann (27c). In dieser Stufe hat man ein System zur Verfügung, das in der Funktionalität vollständig dem hier vorgestellten ODMG-System entspricht. Ein Beispiel für die direkte Objektintegration in einer CORBA-Umgebung und weitere Details zur Umsetzung sind in [RBPF99] zu finden.



a) direkter Zugriff auf Einzelobjekte

b) deklarativer Zugriff auf Originalobjekte

c) voller deklarativer Zugriff

Bild 27: Schrittweise Erweiterung um Anfragefunktionalität

7 Verwandte Arbeiten

In diesem Abschnitt stelle ich weitere Arbeiten im Bereich objektorientierter Sichten vor. Die ersten beiden Ansätze sind dabei von besonderer Bedeutung für die vorliegende Arbeit. Der Ansatz von *Dobrovnik und Eder* stellt eine Alternative für Sichten im ODMG-Standard vor, und die Objekterzeugung im Ansatz von *Heiler und Zdonik* bildete den Ausgangspunkt für das vorliegende Sichtsystem. Danach werden weitere wichtige Arbeiten detailliert beschrieben. Ein Vergleich der unterschiedlichen Ansätze schließt das Kapitel.

Bei den vorgestellten Arbeiten handelt es sich stets um theoretische Arbeiten oder Forschungsprototypen. Kommerzielle objektorientierte Datenbanksysteme bieten entweder nur rudimentäre Sichtfunktionen (wie zum Beispiel die Objektselektion in Ontos oder tupelbasierte Anfragen) oder unterstützen überhaupt keine Sichten (vgl. [Mot96]). Die Forschungsarbeiten selbst konzentrieren sich teilweise auf völlig unterschiedliche Aspekte oder treffen unterschiedliche Annahmen über die Eigenschaften der Sichten.

> *„OODBMS products do not typically provide such a view facility. There have been several proposals in the literature, but there is little agreement here on how a view mechanism should operate.“ [Zdo93, S. 10]*

Daran hat sich auch in den vergangenen Jahren nichts geändert; brauchbare Sichtfunktionalität ist in kommerziellen Systemen noch immer nicht zu finden.

7.1 ODMG-Sichten von Dobrovnik und Eder: eXoT/C

Michael Dobrovnik und Johann Eder von der Universität Klagenfurt arbeiten seit 1993 an einem Sichtmodell, das auf ODMG–93 anwendbar ist. Ihr Ansatz ist jedoch rein konzeptuell und zieht die Probleme der ODMG-Sprachbindung nicht in Betracht. In [DoE93a] wird das Grundkonzept vorgestellt, in [DoE93b] folgt die verwendete Syntax zur Typableitung. Eine Übersicht findet man in [DoE96]. Eine umfassende Beschreibung mit formalen Definitionen und Beweisen erfolgt in [DoE97]; dies ist die umfangreichste Veröffentlichung zur Dissertation Michael Dobrovniks [Dob95].

Der vorgestellte Ansatz basiert auf einer objekterhaltenden Definition externer Schemata. Jedes Schema ist in sich abgeschlossen und besteht aus einer Menge abgeleiteter Klassen mit möglicher Vererbungshierarchie. Jede einzelne Klasse wird dabei von genau einer Basisklasse abgeleitet. Die Ableitung besteht dabei aus vier Schritten:

1. *Projektion:*
   Eine Projektionsliste definiert alle Attribute und Methoden, die von der Basisklasse übernommen werden sollen. Alle anderen Attribute und Methoden sind in der Sicht nicht verfügbar; sie können allerdings zur Methodenimplementierung innerhalb der Sicht verwendet werden.
2. *Erweiterung:*
   Neue Methoden können hinzugefügt werden. Neue Attribute werden nicht unterstützt.
3. *Anpassung:*
   Um die Abgeschlossenheit der Sicht zu gewährleisten, müssen in Attributen und Methodensignaturen alle Typen des Basisschemas durch Typen der Sicht ersetzt werden. Hierbei darf teilweise auch eine Spezialisierung der Parameter stattfinden.
4. *Eingliederung:*
   Der abgeleitete Typ wird in den Typverband der Sicht eingegliedert; hierbei wird mehrfache Vererbung unterstützt.

Der Ansatz verfolgt dabei eine strenge Trennung von konzeptuellem und externem Schema. Für jede Sicht wird ein eigener, unabhängiger Typverband angelegt, der nicht mit dem Originalverband in Beziehung steht und auch keine der Originalklassen enthalten darf. Die Sichtklassen werden dann direkt mit den Originalinstanzen der Datenbank gefüllt (objekterhaltende Sicht). Im allgemeinen wird dabei von einer impliziten 1-zu-1-Abbildung der Extents ausgegangen: alle Instanzen einer Datenbankklasse gehören automatisch zum Extent aller Sichtklassen, die von dieser Klasse abgeleitet sind. Nur in [Dob95] wird näher auf eine anfragebasierte Instanziierung eingegangen.

Durch den objekterhaltenden Ansatz sind die Sichten grundsätzlich update-fähig, da stets auf den ursprünglichen Datenbankobjekten gearbeitet wird. Ein Problem ergibt sich jedoch beim Bestimmen von Methoden-Implementierungen. Aus den Beschreibungen kann man entnehmen, daß der Ansatz keine mehrfache Klassenzugehörigkeit von Datenbankobjekten fordert, sondern das Sichtsystem als Hülle um die Datenbank legt. Das System arbeitet dann zwar mit den Originalobjekten, hat aber ein eigenes Dispatch-Verfahren („*fetch*“), um kontextabhängig Sichtmethoden oder Originalmethoden auf diesen Objekten aufzurufen. Dieses Verfahren verwendet die Methodensignatur, den statischen Sichtkontext des Objekts und den realen Typ des Datenbankobjekts. Es ist jedoch nicht möglich, die spezifischste Subklasse innerhalb der Sicht zu bestimmen, zu dem das Objekt gehört. Daher muß durch eine Reihe von Forderungen sichergestellt werden, daß es trotz dieser fehlenden Information nicht zu Mehrdeutigkeiten kommt. Durch „*well-formed derivation*“ wird zugesichert, daß alle Attribute und Signaturen in der kombinierten Typhierarchie der sogenannten *kovarianten Vererbung* entsprechen: Attribute und Parameter dürfen in einem Subtyp spezialisiert, aber nicht generalisiert oder beliebig verändert werden. Zusätzlich sind drei globale Schema-Invarianten notwendig, die Inkonsistenzen bei Mehrfachvererbung oder Abbildung einer Datenbankklasse auf mehrere Sichtklassen vermeiden.

Das Sichtsystem basiert nicht auf ODMG, kann aber darauf angewandt werden ([DoE94]). Mögliche Erweiterungen in Bezug auf objektgenerierende Klassen werden in [Dob95] untersucht. Zwei weiterführende Artikel zeigen die Verwendung des Sichtsystems zur Datenbankintegration und -replikation ([EdF94], [DoE98]).

Vergleich: Der Ansatz von Dobrovnik und Eder stellt einen guten und schlüssigen Ansatz zur Erweiterung objektorientierter Datenbanken um externe Schemata da. Leider entsteht beim Lesen der Eindruck, daß er durch die mehrfache Klassenzugehörigkeit der Sichtobjekte im Widerspruch zu den Eigenschaften von ODMG–93 steht. Erst durch Vergleich und Analyse der Veröffentlichungen wird deutlich, daß dies nicht der Fall ist und die Zugehörigkeit zu Sichttypen nur simuliert wird. Diese Simulation ist jedoch im Rahmen der Sprachbindung von ODMG nicht möglich, da jedes Datenbank- und Sichtobjekt auf ein C++-Objekt abgebildet wird, bei dem eine solche dynamische Typzuordnung nicht mehr möglich ist. Durch Einführung einer zusätzlichen Indirektion in der Sprachbindung könnte dies zwar erreicht werden, eine Untersuchung zu diesem Thema existiert jedoch nicht. Eine Implementierung ist bisher nicht erfolgt, ist aber für das Datenbanksystem Carinthia geplant. Objektgenerierende Sichten, die für komplexere Abbildungen notwendig sind, werden nur am Rande untersucht. Der Zugriff auf externe Objekte ist ebenfalls nicht untersucht worden.

7.2 Objektsichten in FUGUE (Heiler und Zdonik)

Die zweite wichtige Arbeit auf diesem Gebiet sind die Objektsichten von Sandra Heiler und Stan Zdonik ([HeZ88b], [HeZ90]). Sie kombinieren bereits objekterhaltende und objektgenerierende Verfahren, um Updates auf ihren Sichten zu gewährleisten, und ihr Ansatz diente zum Teil als Ausgangspunkt meiner eigenen Arbeit.

Die Sichten basieren auf dem funktionalen Objektmodell FUGUE[35] ([HeZ88a], [HeZ88b]) und der Objektalgebra von ENCORE ([ShZ89], [ShZ90], [ZdM91]). FUGUE definiert Objekte, Funktionen und Typen. Ein *Objekt* repräsentiert wie üblich eine abstrakte Entität und wird stets über eine Oid adressiert. Das Objekt selbst besitzt weder eine sichtbare Struktur noch Verhalten; beides wird gemäß dem funktionalen Datenmodell durch *Funktionen* hinzugefügt ([Shi81]). Ein *Typ* definiert dabei, welche Funktionen auf ein bestimmtes Objekt anwendbar sind. Zusätzlich existieren Konstruktoren für parametrisierte Typen, mit deren Hilfe neue Mengen {T} und Tupel [T_1, T_2, ...] erzeugt werden können. FUGUE selbst legt nicht fest, wie Objekte oder Funktionen implementiert werden. Es geht davon aus, daß ein Objektmanagementsystem (OMS) zur Verfügung steht, das die notwendige Funktionalität zur Verfügung stellt. Das OMS ist dafür zuständig, für eine gegebene Oid das zugehörige Objekt zu finden und die gewünschte Funktion darauf auszuführen. Alle Funktionen werden als Methoden in der Programmiersprache des OMS realisiert; FUGUE sieht keine eigene Sprache hierfür vor. Außerdem soll das OMS bereits in der Lage sein, auf heterogenen und verteilten Objekten zu arbeiten. Hierzu wird, ähnlich wie bei meinen GOIDs, jede Oid in zwei Teile zerlegt: eine Typinformation, die Herkunft und Art des Objekts beschreibt, und eine interne Id, die von jedem Quellsystem unabhängig vergeben werden kann ([HeZ90], [HeB88]).

Die in FUGUE verwendete Anfragesprache basiert auf der Objektalgebra von ENCORE. ENCORE verwendet den Lambda-Kalkül, um eine Reihe von Operatoren zu implementieren, die denen der relationalen Algebra entsprechen ([ShZ90]):

- **Select**(S, λo C(o)) liefert alle Datenbankobjekte o der Menge S, die die Bedingung C(o) erfüllen.
- **Image**(S, λo F(o)) wendet die Funktion F auf jedes Objekt o der Menge S an und liefert die Menge aller Ergebnisse.
- **Project**(S, λo <(N_1,F_1(o)), ..., (N_n,F_n(o))>) wendet die Funktionen F_1 bis F_n auf jedes Objekt o der Menge S an und liefert die Menge der so konstruierten Ergebnistupel. N_1 bis N_n sind dabei die Namen der einzelnen Tupelelemente (Attribute).
- **Ojoin**(A, B, N_a, N_b, λaλb C(a,b)) bildet das Kartesische Produkt der Objektmengen A und B und liefert all jene Paare als Tupel mit den Attributnamen N_a und N_b zurück, die die Bedingung C(a,b) erfüllen. Falls A bereits eine Menge von Tupeln darstellt, werden keine neuen Paare zurückgeliefert, sondern das jeweilige B-Objekt als neues Attribut an die vorhandenen Tupel aus A angehängt (bzw. an Kopien dieser Tupel). Entsprechend wird verfahren, wenn B oder A und B Tupel enthalten. Diese automatische Entnestung gewährleistet Assoziativität der Ojoin-Operation.

Darüber hinaus gibt es eine Reihe von Mengenoperatoren für die Arbeit mit geschachtelten Mengen und Tupeln (**Union**, **Difference**, **Intersection**, **Flatten**, **Nest**, **UnNest**) sowie den üblichen Punkt-Operator für den Zugriff auf Tupel-Attribute.

Die Algebra sieht keine Funktionen vor, um neue Objekte zu konstruieren; alle Anfragen liefern stets originale Datenbankobjekte oder aus ihnen konstruierte Mengen und Tupel zurück. Im Gegensatz zu der meist üblichen wertorientierten Mengenauswertung haben Mengen und Tupel in ENCORE ebenfalls eine Identität. Hierdurch kann es vorkommen, daß eine Ergebnismenge mehrere Elemente mit identischen Werten aber unterschiedlicher Identität enthält. Um auch wertorientierte Auswertungen zu ermöglichen, definiert ENCORE verschiedene Äquivalenzkriterien sowie zwei Umwandlungsroutinen zur Wertebereinigung:

35. FUGUE = Function Generalization in a Uniform Environment

- *0-equality ($=_0$)* ist das Standardvergleichsverfahren und testet die Identität von Objekten.
- *i-equality($=_i$), i>0* testet Wertegleichheit bis zu einer gegebenen Tiefe i. Zwei Mengen sind i-equal, wenn ihre Elemente paarweise (i-1)-equal sind; zwei Tupel sind i-equal, wenn sie denselben Typ haben und ihre Attribute (i-1)-equal sind. Bei i=1 spricht man auch von *shallow equality ($=_s$),* bei i>1 von *deep equality ($=_d$).*
- *id-equality* fordert sowohl Wertegleichheit als auch strukturelle Gleichheit. Zwei Objekte sind id-equal der Tiefe i, wenn sie sowohl i-equal als auch in ihrer Graphdarstellung isomorph sind (siehe [ShZ89], S. 109). Speziell bedeutet dies, daß mehrfach auftretende, identische Teilobjekte im ersten Objekt auch im zweiten Objekt identisch und nicht nur gleich sind. Dies sichert gleiches Verhalten beider Objekte unter Updates zu.
- *DupEliminate(S, $\lambda s_1 \lambda s_2$ eq(s_1,s_2))* entfernt Duplikate aus einer Menge. Der Operator liefert eine neue Menge, die von jeweils allen Elementen der Menge S, die unter dem gegebenen Gleichheitsoperator eq gleich sind, nur jeweils ein (zufällig ausgewähltes) Element enthält. Bei eq kann es sich wahlweise um i-equality, id-equality oder eine beliebige benutzerdefinierte Vergleichsfunktion handeln.
- *Coalesce(S, A, $\lambda s_1 \lambda s_2$ eq(s_1.A,s_2.A))* führt eine ähnliche Operation auf einer Menge von Tupeln aus, indem es für alle Tupel, deren Attribut A unter dem Gleichheitsoperator eq gleich ist, Kopien erstellt und dieses Attribut auf einen (zufällig ausgewählten) Repräsentanten dieser Äquivalenzklasse setzt.

Das in [HeZ88b] und [HeZ90] vorgestellte Sichtsystem erweitert FUGUE um einen Sichtbegriff. Eine Sicht besteht dabei aus einer Menge von Objektmengen und einer Menge von zugeordneten Typen, die in sich abgeschlossen sind.

Beispiel 7.1:

Die folgende Definition erzeugt eine Sicht V, die einen (virtuellen) Typ SeniorEmp enthält. Zum Extent dieses Typs gehören alle Instanzen einer gegebenen Emps-Klasse, deren Gehalt größer ist als 50K. Die Funktion Create_SeniorEmp erzeugt neue Sichtobjekte, die Funktionen Get_Name und Set_Name werden auf entsprechende Funktionen der Originalklasse Emps abgebildet, und Assign_Parking_Space ist eine neue Funktion für die erzeugte Sichtklasse.

```
View V = { (SeniorEmp, Reps: Select(Emps, lambda(e) e.Salary>50K))}
Type SeniorEmp (in V)
   Rep → Emp
   {  Create_SeniorEmp() → S: SeniorEmp,
      Set_Name(E: SeniorEmp, S: String),
      Get_Name(E: SeniorEmp) → S: String,
      Assign_Parking_Space(S: SeniorEmp) → I: Integer }
```

■

Jedes Sichtobjekt erhält eine sogenannte *Repräsentation,* die diejenigen Basisobjekte enthält, von denen das Sichtobjekt abgeleitet worden ist. Die Zuordnung zwischen der Sicht und der Datenbank, also zwischen den Sichtinstanzen und deren Repräsentationen wird automatisch vom System verwaltet. Es wird dabei nicht festgelegt, wie diese Zuordnung realisiert wird, oder wann neue Oids erzeugt oder alte Oids wiederverwendet werden. Alle Funktionen werden in einer externen Programmiersprache implementiert, wobei aber das OMS dafür sorgen muß, daß während der Ausführung geeignete Kontextwechsel durchgeführt werden. Für eine geeignete Parameteranpassung bei diesen Kontextwechseln muß dabei der Programmierer selbst sorgen.

Vergleich: Die Objektsichten in FUGUE können als Grundansatz für mein Sichtsystem bezeichnet werden. Sie definieren bereits virtuelle Typen, verwenden Anfragen, um die Sichten zu fül-

len, sowie Repräsentationen, die meinen Orig-Referenzen entsprechen. Die Veröffentlichung beschränkt sich dabei aber auf einige Konzepte und läßt viele Dinge im Unklaren oder führt sie nur anhand von Beispielen ein. Die gesamte Zuordnung zwischen Sichtobjekten und Datenbankobjekten sowie die notwendigen Kontextwechsel werden automatisch vom darunter liegenden System übernommen, ohne daß näher darauf eingegangen wird, wie dies geschieht. Die durchgängige Verwendung von Oids, die noch dazu als Instanzen unterschiedlicher Typen verwendet werden können, macht das Sichtsystem für ODMG und vergleichbare Datenbankabbildungen nicht anwendbar.

Mein System ist sowohl konzeptuell als auch aus Implementierungssicht reichhaltiger. Heiler und Zdonik unterstützen weder Vererbung noch Relationships und unterscheiden auch nicht zwischen Import- und Konvertierungsklassen. Sie gehen weder auf die Probleme bei vollständiger oder partieller Instanziierung ein, noch auf referentielle Integrität oder die wechselseitige Zuordnung von Sicht- und Originalobjekten. Besonders auffallend ist weiterhin, daß alle Methoden vollständig ausprogrammiert werden müssen und nicht deklarativ hergeleitet werden können. Dies führt zu einem erhöhten Aufwand bei der Sichterstellung und bietet keine Möglichkeit für die in meinem System mögliche Anfrageoptimierung. Außerdem existiert keine ad-hoc Anfragesprache, die vom System unterstützt werden müßte, während andererseits das funktionale Datenmodell wesentlich mehr Basisfunktionalität anbietet als C++. So werden zum Beispiel Typen und Funktionen selbst wieder als Objekte dargestellt, die dynamisch abgefragt und verändert werden können. Ein Objekt kann gleichzeitig zu mehreren Typen gehören und diese Zuordnung auch verändern ([HeZ88b]).

7.3 Datenbankintegration in MIND (Dogaç et al.)

Auf das Datenbank-Integrationssystem MIND (METU Interoperable DBMS) der Universität Ankara bin ich bereits im Zusammenhang mit der CORBA-Portierung meines Systems eingegangen (siehe Abschnitt 6.7.5, S. 128). MIND verwendet die CORBA-Umgebung für einen integrierten, grobgranularen Zugriff auf heterogene Datenquellen. Jede zu integrierende Datenquelle exportiert dabei sein Schema im ODMG-Format sowie ein einzelnes Datenbankobjekt, das Anfragen auf der Datenquelle ausführt und die Ergebnisse mit Hilfe eines Object Request Brokers an das integrierte System zurückgibt. Die Spezifikation des integrierten Schemas basiert dabei auf der von mir in [BFN94] veröffentlichten Syntax zur separaten Sichtspezifikation. Im Gegensatz zu meiner Arbeit verwendet MIND virtuelle Klassen ausschließlich zur Integration, stellt also kein eigenständiges Sichtsystem für eine einzelne Datenbank zur Verfügung. Außerdem erfolgt eine rein strukturelle Integration, die nur auf Attribute und Relationships, nicht aber auf Methoden eingeht. Hierdurch kann der prozedurale Zugriff entfallen und das System rein deklarativ mit Anfragezerlegung und -expansion arbeiten. Eine allgemeine Systembeschreibung findet man in [DDÖ98], [KOD*95] und [DDK*96], die Sichtspezifikation in [DDK*96] und [NKOD96] und eine detaillierte Beschreibung der Anfragebearbeitung in [NKOD96] und [ONK*96].

7.4 Sichten und Integration in COCOON (Scholl et al.)

Ein weiteres bekanntes Sichtsystem existiert für den Datenbank-Prototyp COCOON der ETH Zürich. Das Objektmodell erweitert einen relationalen NF^2-Datenbankkern um objektorientierte Eigenschaften wie Objektidentität, Funktionen und Vererbung ([ScS90], [ScS92]). Es basiert auf den folgenden Konzepten:

- Ein *Objekt* ist eine Instanz eines abstrakten Typs mit eigener Identität.
- Ein *(abstrakter) Typ* ist eine benannte Menge von *Funktionen*, die auf die Instanz-Objekte dieses Typs angewandt werden können.
- *Daten* sind Werte konkreter Typen, wie etwa Zahlen, Zeichenketten oder Tupel.
- Eine *Klasse* ist eine benannte Menge von Objekten, die alle einem bestimmten, der Klasse zugeordneten Typ entsprechen müssen. Zusätzlich können explizite Bedingungen (*class predicates*) angegeben werden, denen jedes Element genügen muß.
- Eine *Sicht* ist eine benannte Anfrage, die eine neue (virtuelle) Klasse oder NF^2-Relation definiert.

Bei den Funktionen wird nicht zwischen Attributen, Relationships und Methoden unterschieden; die interne Struktur der Objekte ist somit nicht Teil des Datenmodells sondern bleibt der Implementierung überlassen.[36] Für jede Funktion kann eine Inverse angegeben werden. Die Typen sind in einem Typverband angeordnet und erben alle Funktionen ihrer Supertypen. Ein Subtyp kann geerbte Funktionen spezialisieren sowie weitere Funktionen definieren. Die Klassen sind in einer Klassenhierarchie angeordnet und enthalten jeweils eine Teilmenge der gemeinsamen Elemente aller Superklassen. In diesem Modell erfolgt somit eine strikte Trennung intensionaler und extensionaler Aspekte. Darüber hinaus kann jedes Objekt gleichzeitig zu mehreren unabhängigen Typen (*multiple instantiation*) und mehreren unabhängigen Klassen (*multiple class membership*) gehören.

Entsprechend dem Modell ist auch die Anfragesprache COOL eine natürliche Erweiterung der relationalen Algebra. Klassen übernehmen die Rolle von Relationen, Funktionen die Rolle von Attributen, und alle Operatoren können orthogonal so angewandt werden, wie es die involvierten Typen erlauben. Die Anfragen sind grundsätzlich objekterhaltend, um auch Updates ausführen zu können. Alle Anfrageergebnisse werden dynamisch sowohl in die Typ- als auch in die Klassenhierarchie eingebettet. Die verwendeten Operatoren lassen sich in drei Gruppen einteilen:

Relationale Operatoren:

- **select**[cond](set) liefert alle Objekte, die die Bedingung cond erfüllen.
- **project**[f_1,...,f_n](set) schränkt den Typ der Objekte auf die gegebene Funktionsliste ein (liefert neuen Supertyp).
- **extend**[f_1:=$expr_1$,...](set) fügt dem Typ der Objekte neue (abgeleitete) Funktionen hinzu (liefert neuen Subtyp).
- set_1 **union** set_2 vereinigt die Objektmengen (liefert gemeinsamen Supertyp).
- set_1 **difference** set_2 bildet die Mengendifferenz (liefert linken Originaltyp).
- set_1 **intersect** set_2 bildet den Durchschnitt der Objektmengen (liefert gemeinsamen Subtyp).
- **pick**(set) liefert das Objekt aus einer einelementigen Menge.
- **extract**[f_1,...,f_n](set) erzeugt ein relationales Ergebnis aus den Funktionswerten der Objekte. Diese Operation ist als einzige nicht objekterhaltend sondern werterzeugend (*value-generating*).

36. Die Definitionen sind widersprüchlich. [SLT91] beispielsweise führt wieder eine deutliche Unterscheidung ein. Fast überall werden Funktionen wie reine Datenfelder behandelt, denen zum Beispiel Werte zugewiesen werden können.

Update-Operatoren:

- **insert**[f_1:=value,...](class) erzeugt ein neues Objekt in der angegebenen Klasse.
- **delete**(objs) löscht die gegebenen Objekte aus der Datenbank.
- **add**[objs](set) fügt die gegebenen Objekte zu einer Menge oder Klasse hinzu.
- **remove**[objs](set) entfernt die gegebenen Objekte aus einer Menge oder Klasse.
- **set**[f:=value](set) ändert in allen Objekten den Wert der Funktion f (lokale Zuweisung).
- var:=expr weist einer Variablen einen Wert zu (globale Zuweisung).

Mengen-Iterator:

- **update**[f](set) wendet eine Funktion auf alle Objekte an.

Weiterhin gibt es noch eine Reihe von Standard-Operatoren für die Basis-Datentypen, und die auf den Typen definierten Funktionen bilden ebenfalls einen Teil der verfügbaren Sprachelemente. Ein objekterhaltender Join kann mit Hilfe von extend ausgedrückt werden, ein objektgenerierender Join wird ausschließlich in [ScS90] erwähnt.

Eine weitere besondere Eigenschaft von COCOON ist, daß Typen, Klassen und Funktionen selbst als Objekte in einem Metaschema vorliegen und so direkt mit der Anfragesprache manipuliert werden können. Die notwendigen Änderungen auf den betroffenen Instanzen werden dabei automatisch vom System durchgeführt. Auf diese Weise bietet COCOON einen sehr eleganten Ansatz zur Schemaevolution ([Tre91], [SST92]).

Eine *Sicht* in COCOON ist wie im relationalen Fall durch eine benannte Anfrage definiert ([ScS91], [SLT91]). In den meisten Fällen handelt es sich auf Grund der objekterhaltenden Operatoren um strukturell veränderte Originalobjekte. Die Sicht definiert somit eine (virtuelle) Klasse mit dazugehörigem Typ, die jeweils konsistent in die vorhandenen Klassen- und Typhierarchien eingebettet werden. Durch die Objekterhaltung sind die so erzeugten Sichten update-fähig, da alle Änderungen direkt auf den Originalobjekten erfolgen. Bei allen Updates kümmert sich das System automatisch darum, die Extents von Datenbankklassen und Sichtklassen mit Hilfe von Klassenprädikaten konsistent zu halten ([ScS91]). Als zusätzliche Variante werden auch relationale Sichten angeboten, die aber in erster Linie dazu dienen, Daten für die Darstellung beim Endbenutzer aufzubereiten ([ScS90]). Da Sichten stets nur einzelne Klassen darstellen, ist für eine konsistente Datenbankpräsentation beim Benutzer eine zusätzliche Definition sogenannter *Subschemas* notwendig ([SLT91]).

In [SST92] wird das beschriebene Sichtsystem für die Integration von Objektbanken verwendet. In einem ersten Schritt werden die externen Datenbankschemas in einen globalen Namensraum importiert, wobei jeder Name von Datenbank DB_i mit dem Suffix @DB_i versehen wird. Zwei globale Typen object@GOB und bottom@GOB bilden zusammen mit den importierten Verbänden einen neuen Typverband.[37] Anfragen, die auf mehrere Klassen zugreifen, werden auf die einzelnen Originaldatenbanken verteilt, wobei Daten aus einer Datenbank als Parameter an die andere übergeben werden können. Mit Hilfe der extend-Funktion können neue Klassen angelegt werden, die Verweise zwischen unterschiedlichen Datenbanken realisieren. Der nächste Schritt besteht aus der Identifikation identischer Realweltobjekte in den einzelnen Datenbanken. Der Benutzer muß dazu für jedes Typpaar (T_i, S_j) aus unterschiedlichen Datenbanken (DB_i, DB_j) eine Funktion sameT_iS_j definieren, die für eine Instanz von T_i das zugehörige Objekt aus S_j liefert.

37. Dies entspricht der disjunkten Vereinigung von S. 35.

Hierdurch wird eine globale Objektidentität definiert, die für alle Operationen auf der globalen Datenbank verwendet wird. Da mit den Schemas auch alle Metaschemas in die globale Datenbank importiert werden, erfolgt die eigentliche Schemaintegration durch Definition von same-Funktionen auf den Klassen Functions, Types und Classes der einzelnen Objektbanken. Notwendige Konvertierungen oder Umbenennungen müssen dabei zuvor in Form neuer Sichten implementiert werden. Die Veröffentlichung gibt leider keinen Hinweis darauf, wie der Objektimport funktioniert; wie man also externe Objekte in neu erzeugte Klassen des globalen Schemas einfügt.

Vergleich: Der in COCOON verfolgte objekterhaltende Ansatz stellt die konzeptuell sauberste Lösung für objektorientierte Sichten dar. Er kann aber nur auf Systemen implementiert werden, die die folgenden, sehr anspruchsvollen Bedingungen erfüllen (vgl. [ScS91]): i) Das System muß eine objekterhaltende Anfragesprache zur Verfügung stellen. ii) Es muß Klassen und Typen getrennt behandeln, da sonst eine Ergebnisklassifikation nicht möglich ist. iii) Es muß sowohl mehrfache Typ- als auch Klassenzugehörigkeit der Objekte unterstützen. iv) Es muß dynamische Komponenten für eine automatische Reklassifikation veränderter Objekte zur Verfügung stellen. Diese Funktionalität wird jedoch weder vom ODMG-Standard noch von anderen existierenden Objektbanken bereitgestellt. Der Ansatz ist somit nur sehr eingeschränkt verwendbar. Ein weiterer Nachteil ist, daß das System ausschließlich die 1-zu-1-Integration identischer Realweltobjekte zuläßt — es gibt keine Möglichkeit zur Darstellung anderer Korrespondenzformen. Außerdem werden Updates von abgeleiteten Funktionen kategorisch ausgeschlossen, obwohl teilweise auf sogenannte set-Funktionen für klassenspezifische Update-Implementierungen hingewiesen wird. Weitere problematische Updates und deren Lösung werden in [ScS91] und [SLT91] diskutiert.

7.5 OSCAR und LIVING IN A LATTICE (Heuer et al.)

Ähnlich wie COCOON basiert auch der objektorientierte Datenbankprototyp OSCAR der TU Clausthal ([HFW90], [Kla92]) auf einem relationalen System. Das Datenmodell EXTREM (*extended relational model*, [HöH91], [Heu89]) basiert auf dem IFO-Modell von Abiteboul und Hull ([AbH87]) und definiert neben Basis-Datentypen Tupel-, Mengen- und Listenkonstruktoren sowie Objektklassen. Als Besonderheit wird bei der Klassenhierarchie zwischen Spezialisierung und Generalisierung unterschieden, so daß nicht nur neue Subklassen sondern auch neue (*freie*) Superklassen definiert werden können. Für die Implementierung wird EXTREM über ein erweitertes ER-Modell in geschachtelte Relationen und schließlich in das relationale Modell übersetzt, so daß die Objektbank auf der relationalen Datenbank IRIS ([Heu87]) aufgesetzt werden kann. Das Modell trennt Objekte von ihrem Zustand: Objekttypen sind rein abstrakt ohne jegliche Struktur, sie bestehen nur aus einer (unsichtbaren) Oid. Über eine *Zustandsfunktion* wird jedem Objekt ein sogenanntes *Objekttupel* zugeordnet, das die Menge der aktuellen Attributwerte beinhaltet.

Die grundlegende Anfragesprache ist eine Objektalgebra, die die Relationenalgebra an das EXTREM-Modell anpaßt. Die Operationen werden in objekterhaltende und objektgenerierende unterteilt. Erstere liefern die veränderten Originalobjekte zurück und können für eine Sichtdefinition verwendet werden, letztere erzeugen temporär neue Objekte, um das Anfrageergebnis darzustellen. Zu den objekterhaltenden Operationen gehören:

- Projektion **PROJ** [a_1,...,a_n] (set),
- Objekterweiterung **EXTEND** [att←expr] (set),
- Selektion **SEL** [cond] (set),

- Restriktion **REST** [att:cond] (set),
- Komponenten-Join[38] set_1 **COMP_JOIN** [att←rel] set_2,
- Umbenennung **REN** [neu←alt,...] (set), **REN** [präfix] (set),
- is-a-Join set_1 **ISA_JOIN** set_2,
- Restrukturierung **CHANGE** [expr] (set),
- und die Mengenoperationen **UNION**, **INTERSECT**, **MINUS**, **EXCLUDE**.

Zu den objektgenerierenden Operationen gehören:

- objektgenerierende Projektion **GPROJ** [a_1,...,a_n] (set),
- Kreuzprodukt set_1 **CROSS** [classname] set_2
- und Objekt-Konstruktion **GENclass** (arg).[39]

Aufbauend auf dieser Algebra werden eine SQL-ähnliche Anfragesprache O^2QL und die regelbasierte Anfragesprache LIVING IN A LATTICE definiert ([HeS93]).

Um Anfrageergebnisse in die Typ- und Klassenhierarchien einzubetten und objekterhaltende Strukturänderungen zu ermöglichen, verwendet OSCAR ein ganz spezielles Cluster-Modell ([HeS91]): Für alle in der Datenbank definierten Objekttypen gilt die klassische Einheit von Typ und Klasse, bei der strukturelle Vererbung und Teilmengenbeziehung der Instanzmengen übereinstimmen. Im Rahmen der Anfrageauswertung wird jede Klasse jedoch als sogenannter *Cluster* aufgefaßt, in den alle Sichten auf Instanzen dieser Klasse eingetragen werden. Hierdurch bleiben reduzierte Objektmengen oder strukturell veränderte Objekte in „semantischer Nähe“ der Originaldaten und Unentscheidbarkeiten bei der Ergebnisklassifikation werden vermieden. Die Originalklasse des Clusters definiert dabei die Menge der möglichen Instanzen aller abgeleiteten Klassen. Hierdurch ergibt sich eine Schachtelung von drei Verbänden: ein globaler *Klassenverband* beschreibt die Abhängigkeit der Cluster untereinander; jeder dieser Cluster enthält ein Produkt aus zwei weiteren Verbänden — einem *Typverband* mit allen im Cluster verwendeten Typen und einem *Instanzenverband* mit allen auftretenden Extents. Ein Klassifikationsmechanismus sortiert jedes Anfrageergebnis in diese Verbände ein ([HeS91], [Bus91]).

Ein konkretes Sichtkonzept wird nicht explizit diskutiert, ist aber als Teil der Regelsprache zu sehen. Bei der Schemadefinition können abgeleitete Klassen definiert werden, indem man ihre Struktur und Einordnung in den Klassenverband angibt. Die Bestimmung der zugehörigen Instanzen und Attributwerte erfolgt unabhängig davon durch eine definierende Regelmenge.

Vergleich: Genau wie COCOON basiert OSCAR auf einem objekterhaltenden und daher schwer umsetzbaren Datenmodell. Der Schwerpunkt liegt dabei ausschließlich auf strukturellen Aspekten — Methoden werden nur am Rande erwähnt. Weiterhin ist OSCAR in erster Linie als interaktives System konzipiert und bietet keine Sprachbindung — sowohl die Anfragen als auch die Methodenimplementierungen werden in speziellen Datenbanksprachen geschrieben und interpretiert. Das Datenmodell mit seinen geschachtelten Verbänden ist äußerst ungewöhnlich und schwer verständlich; Sichten werden nur als Seiteneffekte der Regelsprache betrachtet. Insgesamt erscheint COCOON wesentlich eleganter und ausgereifter.

---

38. Der Komponenten-Join ersetzt ein objektwertiges Attribut (also eine Relationship) durch das Tupel der Attributwerte des Zielobjekts. Er repräsentiert somit Pfadausdrücke in der Objektalgebra.

39. In [HFW90] werden teilweise andere Operatoren und Bezeichner verwendet (Subtyp-Join **SUB_JOIN**, γ-Projektion PROJ, γ-Rename **REN**, γ-Join **JOIN**, γ-Restrukturierung). Weiterhin gibt es noch objekterhaltende und objektgenerierende Nestung und Entnestung (**NEST**, **UNNEST**).

7.6 Virtuelle Klassen von Abiteboul et al.

Die Objektsichten von Serge Abiteboul und Anthony Bonner von INRIA ([AbB91]) basieren auf dem Datenmodell von O_2 und kombinieren den objekterhaltenden und den objektgenerierenden Ansatz miteinander. Eine Sicht ist hier ein separates Schema, das alle seine Daten aus einer Datenbank importiert und um virtuelle Klassen ergänzt. Eine Sicht wird in vier Schritten aufgebaut:

1. Import der benötigten Klassen und ihrer Subklassen
2. Verstecken unbenutzter Teile in den importierten Klassen (*Hiding*)
3. Modifikation der importierten Klassen und Definition virtueller Klassen
4. Verstecken unerwünschter Attribute

Die ersten beiden Schritte stellen die Basisdaten aus einer oder mehreren Datenbanken zusammen. Diese werden dann im weiteren Verlauf verändert und erweitert:

```
create view My_View;
import all classes from database Chrysler;
import class Person from database Ford;
hide attribute Salary in class Employee.
```

Nach dem Import werden neue virtuelle Klassen erzeugt und in die Hierarchie eingefügt. Hierzu gibt es vier Operationen:

1. *Spezialisierung*
   Die neue virtuelle Klasse wird als Subklasse unter einer existierenden Klasse eingefügt, der Extent ergibt sich aus einem Klassenprädikat:

   ```
   class Adult includes (select P from Person where P.Age > 20).
   ```

2. *Generalisierung*
   Die neue virtuelle Klasse ist gemeinsame Superklasse einer Reihe existierender Klassen. Der Extent ist die Vereinigung aller Extents:

   ```
   class Ship includes Tanker, Cruiser, Trawler.
   ```

3. *Verhaltensgeneralisierung (Behavioral Generalization)*
   Die neue Klasse ist gemeinsame Superklasse all jener Klassen, die einen angegebenen Typ implementieren (strukturelle Subtypen). Die Hierarchie wird automatisch angepaßt, wenn durch Schemaevolution neue Klassen diese Bedingung erfüllen bzw. bisherige Subklassen dies nicht mehr tun:

   ```
   class Printable_Pattern has attribute preview of type Bitmap;
   class Printable includes like Printable_Pattern.
   ```

4. *Erzeugung imaginärer Objekte*
   Während die ersten drei Operationen objekterhaltend sind, stellt die Erzeugung imaginärer Objekte objektgenerierende Funktionalität zur Darstellung von Joins oder anderen größeren Umstrukturierungen zur Verfügung. Der Extent einer solchen *imaginären Klasse* wird hierbei durch eine relationale Anfrage bestimmt. Für jedes Tupel im Ergebnis wird ein neues Objekt erzeugt, das die Tupelwerte als Attributwerte (*core attributes*) hat. Durch eine geeignete Abbildung wird sichergestellt, daß für ein bestimmtes Tupel immer dieselbe Oid verwendet wird:[40]

   ```
   class Family includes imaginary
   (select [Husband:H, Wife:H.Spouse] from H in Person where H.Sex = 'male').
   ```

40. Siehe auch [AbK98] für umfangreiche Diskussionen zur Objektidentität.

Alle so erzeugten Klassen werden in den Verband der importierten Klassen eingebettet, indem automatisch alle herleitbaren isa-Beziehungen bestimmt werden und hieraus die genaue Position ermittelt wird. Zusätzlich erfolgt eine automatische Aufwärts- und Abwärtsvererbung, die die Klasse mit allen gemeinsamen Attributen ihrer Sub- bzw. Superklassen versieht. Weiterhin ist es möglich, die Operationen miteinander zu kombinieren, so daß zum Beispiel eine virtuelle Klasse definiert werden kann, die sowohl vorhandene Klassen als auch Spezialisierungen generalisiert.

Ein besonders interessantes und mächtiges Konzept ist die Definition parametrisierter Sichtklassen. Durch die Angabe eines Parameters können mit einer einzelnen Definition ganze Mengen von Sichtklassen definiert werden, die je nach Bedarf aktiviert werden:

class Adult(A) **includes** (**select** P **from** Person **where** P.Age > A).

Alle virtuellen Klassen können nach ihrer Erzeugung wie normale Datenbankklassen bearbeitet werden. Unter anderem können neue virtuelle Attribute hinzugefügt werden:

attribute Address **in class** Person **has value**
[City: **self**.City, Street: **self**.Street, Zip_Code: **self**.Zip_Code].

Auf Grund dieser berechneten Attribute wird die Unterscheidung zwischen Attributen und Methoden aufgehoben und alles als Methode betrachtet; allerdings wird jede Methode als (gespeichertes oder berechnetes) Attribut interpretiert — auf allgemeine Methoden mit Parameterlisten und Seiteneffekten wird nicht eingegangen.

Beim letzten Schritt der Sichtdefinition werden schließlich die Attribute aus der Sicht entfernt, die durch virtuelle Attribute ersetzt worden sind:

hide attributes City, Street, Zip_Code **in class** Person.

Das verwendete Objektmodell unterstützt Mehrfachvererbung und mehrfache (virtuelle) Klassenzugehörigkeit, jedes Objekt muß jedoch in genau einer spezifischsten nicht-virtuellen Klasse definiert sein, um eine eindeutige Speicherdarstellung zu gewährleisten (*unique root*). Zwischen Typen und Klassen wird nicht unterschieden. Um Typkonflikte durch das Ausblenden von Attributen zu vermeiden, wird in [SAD94] eine spezielle *may_be*-Beziehung definiert, die die Standardvererbung zwischen Datenbankklassen und virtuellen Klassen ersetzt. Außerdem wird dort kurz auf die Anwendung dieser Sichten zur Datenbankintegration eingegangen.

Die kommerzielle Datenbank O_2 von O_2 Technology bzw. Ardent Software bildet die Grundlage für die vorgestellten Arbeiten und bietet selbst keine Sichtfunktionalität. Es gibt weder virtuelle Klassen noch mehrfache Klassenzugehörigkeit; eine gewisse Flexibilität wird allerdings dadurch angeboten, daß man einzelnen Objekten dynamisch neue Attribute und Methoden hinzufügen kann (*exceptional attributes/methods*). Eine sehr umfangreiche Beschreibung der Datenbank und ihrer Konzepte findet man in [BDK92]; eine Weiterentwicklung in Richtung ODMG wird in [Ala99] untersucht.

Vergleich: Durch den vorwiegend objekterhaltenden Ansatz ist dieses Sichtmodell mit dem von COCOON vergleichbar; es verwendet jedoch ein wesentlich einfacheres Datenmodell. Hierdurch ist es zwar nicht ganz so mächtig, kann jedoch besser in andere Systeme integriert werden. Die im objektgenerierenden Teil verwendeten core attributes entsprechen grundsätzlich dem von mir verwendeten orig-Attribut, wie auch die Diskussion über berechnete Oid-Erzeugung zeigt. Im Gegensatz zu meinem Ansatz ist die Funktionalität jedoch sehr eingeschränkt: Imaginäre Klassen können nie in einer Vererbungshierarchie stehen, alle Herleitungen sind ausschließlich deklarativ, es werden keine allgemeinen Methoden unterstützt, und Updates werden ebenfalls kaum angesprochen — abgesehen von der Einschränkung, daß der Anwender grund-

sätzlich keine neuen virtuellen Objekte erzeugen kann. Auch auf die technische Realisierung des objekterhaltenden Imports, einer zentralen Voraussetzung der mehrfachen Klassenzugehörigkeit, wird nicht näher eingegangen, sondern auf private Korrespondenz verwiesen.

7.7 Pegasus, Iris und OpenODB (Kent et al.)

Ein weiteres objektorientiertes System zum Zugriff auf verteilte Datenbanken ist das *Pegasus*-System der Hewlett-Packard Laboratories ([AAD*93], [KAA*92], [SAD*95]). Es kombiniert und erweitert die Systeme *Iris* ([FBA*90]) und *OpenODB* zu einem Informations-Management-System mit integriertem Zugriff auf heterogene Informationsquellen. Genau wie in COCOON und FUGUE wird das funktionale Datenmodell verwendet. *Objekte* sind an sich strukturlos und bekommen ihre Eigenschaften durch Zuordnung zu einem Typ; ein *Typ* definiert die Menge der Funktionen, die auf die Objekte seines Extents angewandt werden können; *Funktionen* realisieren Attribute, Relationships und Methoden. Im Gegensatz zu COCOON definiert Pegasus eine 1-zu-1-Zuordnung von Klassen und Typen, so daß Strukturvererbung und Inklusionsabhängigkeit zusammenfallen. Ein Objekt kann gleichzeitig zu mehreren Typen gehören und diese Typen auch dynamisch wechseln. Auch Schemaevolution ist zur Laufzeit möglich, da alle Typen selbst wieder Objekte sind, die mit der Anfragesprache HOSQL (Heterogeneous Object SQL) manipuliert werden können ([AAC*95], [SAD*95]).

Pegasus bietet kein explizites Sichtsystem sondern konzentriert sich auf die Probleme bei der Integration unterschiedlicher Quellen. Für jede externe Quelle werden sogenannte *Producer Types* definiert. Ein solcher Typ stellt die Objekte und Funktionen zur Verfügung, über die auf die externen Daten zugegriffen wird. Über eine relationale Anfrage, die sogenannte *Producing Expression*, wird eine Schlüsselmenge bestimmt, aus der dann die zugehörigen Oids berechnet werden. Die Oids können dabei durch geeignete Strukturierung an die unterschiedlichen Datenquellen und eventuelle Vererbungshierarchien angepaßt werden ([KAA*92]). Über eine *Identifying Function* kann für jedes importierte Objekt der zugehörige Schlüsselwert bestimmt werden:

```
CREATE PRODUCER TYPE EDB.Student
   FROM EDB
   PRODUCING BY SQL(SELECT StudId FROM Students)
   FUNCTIONS (
      StudentId Integer AS IDENTIFIER;
      SSNum Integer AS MAP TO Students.SSNo;    );
```

Ein Administrator kann nun importierte Objekte verschiedener Quellen integrieren. Hierzu definiert er zwei Funktionen: Unifier(T) liefert für einen importierten Typ T den zugehörigen integrierten Typ; Image(x) liefert zu einem Objekt x∈T das zugehörige integrierte Objekt in Unifier(T). Hieraus ergibt sich die Einschränkung, daß ein externes Objekt stets nur an einer Integration teilnehmen kann, und daß ausschließlich 1-zu-1-Unifikation unterstützt wird.

```
CREATE FUNCTION Image(EDB.Student y) → Student z
   AS HOSQL SELECT z WHERE ssnum(z)=ssnum(y);
```

Die Deklaration von importierten und integrierten Typen variiert dabei in den verschiedenen Veröffentlichungen. In [SAD*95] beispielsweise wird für die Integration ein sogenannter *Covering Supertype* über den zu integrierenden Typen eingefügt, der mittels *Upward Inheritance* die zugehörigen Funktionen erbt. Die Unifikation auf Instanzebene erfolgt durch Definition einer Äquivalenzbedingung:

```
CREATE TYPE Employee
   AS COVERING SUPERTYPE OF Programmer, Engineer;
DEFINE OBJECT IDENTITY ON (Programmer p | Engineer e)
   BY Programmer.Ssno(p)=Engineer.Ssno(e);
```

Für die Beseitigung struktureller Inkonsistenzen werden vordefinierte Abgleichsfunktionen (*Reconciling Functions*) zur Verfügung gestellt. Man kann dadurch wählen, ob alle Daten behalten werden, ob bestimmte bevorzugt werden, oder ob Summen oder Mittelwerte berechnet werden sollen ([SAD*95]):

```
DEFINE RECONCILING FUNCTION Salary (Programmer | Engineer) → REAL
DISAMBIGUATE DISAMB_SUM;
```

Für die direkte Integration unterschiedlicher Quellen wird in [KAA*92] noch die Möglichkeit einer globalen Oid-Ersetzung diskutiert, bei der alle zu unifizierenden Objekte nachträglich dieselbe Oid zugewiesen bekommen. Eine andere Alternative sind die mengenbasierten Oids von Chen und Shan, auf die ich weiter unten noch eingehe. Die Definition von integrierten Schemata erfolgt mit Hilfe von Zugriffsrechten. Nachdem die Integration vollständig spezifiziert ist, werden die Originalklassen vor dem Endbenutzer verborgen, während dem Administrator stets alle Klassen zur Verfügung stehen.

Vergleich: Pegasus bietet kein reines Sichtsystem sondern ist ausschließlich für die Integration heterogener Informationsquellen gedacht. Die Möglichkeiten für strukturelle Veränderungen scheinen sehr beschränkt zu sein, und die verwendeten Datenquellen sind ausschließlich relational — Objekte werden erst in der integrierten Ebene verwendet, um durch Vererbung, Objektidentität und berechnete Attribute die Daten besser verknüpfen zu können. Darüber hinaus wird mit dem funktionalen Modell wieder ein sehr mächtiges Objektmodell vorausgesetzt. Das System hat sich im Laufe der Veröffentlichungen offenbar stark verändert, so daß es schwer ist, sich ein geschlossenes Bild davon zu machen, welche der vorgestellten Eigenschaften und Komponenten letztendlich umgesetzt worden sind.

Multidatabase Mappings von Chomicki und Litwin

Eine andere, eher formale Erweiterung von IRIS wird von Chomicki und Litwin in [ChL92] vorgestellt. Auch sie definieren kein reines Sichtsystem sondern konzentrieren sich direkt auf die Integration, wobei aber die Beseitigung struktureller Inkonsistenzen im Vordergrund steht. Der Grundansatz ist zunächst, daß neue Typen mit einfachen Selektions- und Vereinigungsanfragen gefüllt werden. Durch die Einführung parametrisierter Typen, sogenannter *Type Schemas*, kann dabei eine einzelne Ausgangsklasse automatisch in viele Zielklassen aufgeteilt werden:

```
CREATE TYPE EmpPos[String p] AS
SELECT e FOR EACH Employee e WHERE Pos(e)=p.
```

Neben objekterhaltenden Typdefinitionen können Typen auch objektgenerierend definiert werden, indem für jedes Tupel einer Anfrage ein neues Objekt erzeugt wird:

```
CREATE TYPE Department PROPERTIES (Name) AS
SELECT DISTINCT d FOR EACH Employee e, String d WHERE Dep(e)=d.
```

Da alle Typen und Funktionen selbst als Objekte vorliegen, können durch Anfragen auf dieser Metaebene neue Typen und Funktionen definiert werden, deren Ursprungstypen nach dynamischen Kriterien variieren können. Weiterhin gibt es parametrisierte Funktionsdefinitionen, wobei aber nicht deutlich wird, inwiefern ein (Meta-)Parameter in der Funktionsdefinition einen Vorteil gegenüber einem zusätzlichen (normalen) Parameter der Funktion darstellt.

Für alle erzeugten Typen gilt die sogenannte *Typkohärenz*: Die Typhierarchien unterschiedlicher Datenbanken sind grundsätzlich voneinander getrennt. Wird ein Typ aus einer anderen Datenbank importiert, so müssen auch alle referenzierten Datentypen importiert werden, um Abgeschlossenheit zu gewährleisten. Die Aussage zur konkreten Typhierarchie bei abgeleiteten Klas-

sen ist widersprüchlich: Einerseits wird vom Benutzer gefordert, eine gewünschte Typbeziehung explizit anzugeben, andererseits wird die Existenz expliziter Typhierarchien aufgrund der dynamischen Effekte parametrisierter Typen ausgeschlossen.

Vergleich: Genau wie Pegasus ist der Ansatz von Chomicki und Litwin auf Integration spezialisiert und fußt auf dem mächtigen funktionalen Modell mit mehrfacher Typzugehörigkeit, dynamischer Typevolution und der Importfunktion von Iris. Im Gegensatz zu Pegasus werden jedoch Objekte und nicht nur relationale Daten integriert, wobei die Herleitungen direkter und daher leichter verständlich sind. Ein ganz entscheidender Nachteil ist jedoch, daß das System ausschließlich für die disjunkte Vereinigung vorgesehen ist — zumindest sind in dem Beitrag keinerlei Unifikationsmöglichkeiten beschrieben.

7.8 MultiView (Rundensteiner et al.)

Ein weiteres objekterhaltendes Sichtsystem ist *MultiView* von Rundensteiner et al. von der University of Michigan ([Run92], [KuR96a], [Run96]). Es folgt der üblichen Einheit von Klasse und Typ und basiert auf der Smalltalk-Datenbank GemStone. Eine Sichtdefinition in MultiView erfolgt dabei in drei Schritten:

1. *Anpassung (customization):*
   Mit Hilfe einer Objektalgebra werden die in der Datenbank vorhandenen Klassen an die eigenen Bedürfnisse angepaßt. Hierzu stehen die folgenden Operatoren zur Verfügung:

   - **hide** entfernt Properties aus einer Klasse und erzeugt eine neue Superklasse:

     OldClass **createHideClass:** NewName **hideMethods:** PropertyList

   - **refine** fügt Properties zu einer Klasse hinzu und definiert eine neue Subklasse:

     OldClass **createRefineClass:** NewName **withProperties:** PropDefList

   - **select** erzeugt eine neue Subklasse mit reduziertem Extent:

     OldClass **createSelectClass:** NewName **query:** Predicate

   - **union** vereinigt die Extents zweier Klassen zu einer neuen Superklasse. Die Struktur enthält alle gemeinsamen Properties (kleinster gemeinsamer Supertyp):

     OldClass **createUnionClassWith:** OtherClass **named:** NewName

   - **intersect** bildet eine neue Subklasse mit allen gemeinsamen Instanzen. Die Struktur enthält sämtliche Properties beider Klassen (größter gemeinsamer Subtyp):

     OldClass **createIntersectClassWith:** OtherClass **named:** NewName

   - **difference** erzeugt eine strukturell gleiche Subklasse mit eingeschränktem Extent (Mengendifferenz). Es stellt somit einen Sonderfall von select dar:

     OldClass **createDifferenceClassWith:** OtherClass **named:** NewName

   - **join** erzeugt eine völlig neue Klasse mit neuen Instanzen, basierend auf der Verknüpfung zweier Basisklassen:

     Class1 **createJoinWith:** Class2 **named:** NewName **withMethods:** MethodList
     joinAtt1: Propname1 **joinAtt2:** Propname2

   Die ersten beiden Operatoren erlauben strukturelle Veränderungen einzelner Klassen, mit select kann der Extent einer Klasse eingeschränkt werden, und die drei Mengenoperatoren definieren neue Sub- oder Superklassen mit gemeinsamen Elementen. Der join-Operator kommt erst in [KuR96a] hinzu und wird nicht näher erläutert. Aus der Parameterliste geht hervor, daß er einen Equi-Join auf zwei Klassen ausführt und für jedes passende Objekt-

paar ein neues Objekt in einer neuen Klasse anlegt. Er ist somit der einzige objektgenerierende Operator der Algebra. Wie er allerdings umgesetzt werden soll, wird nicht beschrieben.

In [Run93] werden die objekterhaltenden Operatoren von Klassen auf vollständige Teilgraphen erweitert. Die Operatoren hide*, **refine***, **select***, **union***, **intersect*** und **difference*** führen die entsprechende Operation nicht nur auf der angegebenen Klasse aus, sondern rekursiv auf allen Subklassen. Diese Graphalgebra scheint allerdings nicht nach Smalltalk umgesetzt worden zu sein.

2. *Integration:*
   Der von den Autoren gewählte Begriff der Integration ist hier irreführend, da es sich nicht um eine Schemaintegration handelt, sondern um die Einordnung der neu erzeugten Klassen in den existierenden Klassenverband. Die Einordnung erfolgt dabei automatisch anhand der Algebra-Operatoren und kann nicht vom Benutzer beeinflußt werden. Für eine korrekte Einordnung muß dabei für jeden einzelnen Operatoraufruf (mindestens) eine neue Klasse erzeugt und klassifiziert werden — es ist nicht möglich, mehrere Operatoren auf einmal anzuwenden ([KuR96a], S. 207). Eine weitere Forderung ist der sogenannte *Single Point of Inheritance*: Wenn zwei Klassen dasselbe Datenfeld besitzen, müssen sie es beide von einer gemeinsamen Superklasse geerbt haben. Auf die Algorithmen zur automatischen Klassifikation sowie auf das dabei auftretende Problem der Unentscheidbarkeit wird nicht näher eingegangen sondern auf Technische Berichte verwiesen (z.B. [Run92b]).
3. *Auswahl (view selection):*
   Das Ergebnis der Integration ist ein globales Schema (GS), bestehend aus dem ursprünglichen Basisschema (BS) und allen neu erzeugten virtuellen Klassen. Für jeden Endbenutzer wird jetzt die Menge der für ihn zugreifbaren Klassen ausgewählt. Um Abgeschlossenheit der Sicht zu gewährleisten, wird von dieser Menge automatisch die transitive Hülle aller erreichbaren Klassen gebildet und mit aufgenommen (*Closed-View-Generation, CVG*). In diesem letzten Schritt werden keine Veränderungen mehr vorgenommen; jedes Sichtschema (*View Schema, VS*) ist eine konsistente Teilmenge desselben globalen Schemas.

Beispiel 7.2:

Das folgende Smalltalk-Fragment erzeugt eine neue Select-Klasse und fügt sie einer Sicht hinzu:

```
run
Person createSelectClass: #YoungPerson query: [ :person | person birthyear > 1975 ].
run
MyView addVC: YoungPerson withName: #MyYoungPerson.
```

■

Der vorgestellte Ansatz soll dabei unabhängig von der verwendeten Algebra sein; allerdings müßte bei anderen Operatoren die automatische Klassifikation entsprechend erweitert werden, was die Freiheit bei der Wahl der Algebra wiederum einschränkt. Durch eine formale Analyse wird in [Run92] die sogenannte *Sichtunabhängigkeit (View Independence)* zugesichert: Die Definition einer neuen Sicht, mit allen drei oben angegebenen Schritten, hat keine sichtbaren Auswirkungen auf bereits vorhandene Sichten. Obwohl neue Klassen hinzugefügt und vorhandene Klassen strukturell verändert werden, bleibt die Funktionalität jeder bereits existieren Klasse unverändert erhalten. Zur Anfrageoptimierung können Sichtklassen auch materialisiert in der Datenbank gehalten werden; Extentänderungen werden in diesem Falle durch Trigger realisiert. In [KuR95] und [KuR96b] werden hierzu weitere Algorithmen angegeben, die über Filter unnötige Updates verhindern, mittels hierarchischer Registrierung einen optimierten Notifikationsmechanismus realisieren und dabei bestimmte Update-Anomalien vermeiden. Eine weitere positi-

ve Eigenschaft ist MultiViews Fähigkeit zur persistenten Sichtergänzung (*Capacity Augmenting View*): Da jede Sichtklasse in den Klassenverband eingeordnet wird und somit in der Datenbank repräsentiert ist, können zu Sichtklassen auch neue Datenfelder hinzugefügt werden, die nicht aus anderen Daten hergeleitet sondern direkt mit beliebigen Werten gefüllt werden. Hierdurch bietet sich MultiView als Grundlage zur Schema-Evolution an, bei der mehrere Versionen eines Schemas parallel zur Verfügung gestellt werden können (*Transparent Schema Evolution (TSE)* [RaR95a], [RaR95b]).

Das vorgestellte Modell stellt dabei sehr hohe Anforderungen an die zugrunde liegende Datenbank. Es fordert mehrfache Klassenzugehörigkeit der Objekte, Mehrfachvererbung, dynamische Klassen- und Objektmodifikationen sowie dynamische Umstrukturierungen in der Vererbungshierarchie. In [KuR96a], S. 204 wird sogar festgestellt, daß kein kommerzielles Datenbanksystem die notwendige Funktionalität zur Verfügung stellt:

> *„Second, and more critically, to the best of our knowledge, no existing commercial OODB system supports all of the properties we identify as critical for the realization of fully functional object-oriented views.“*

Selbst das von MultiView verwendete GemStone ist nicht direkt verwendbar. Um die notwendige Funktionalität zur Verfügung zu stellen, wird eine völlig neue Objektrepräsentationsschicht um den Datenbankkern herumgelegt ([KuR96a]). In der sogenannten *Object-Slicing Representation* wird jedes Objekt durch eine Menge von Datenbankobjekten dargestellt, bestehend aus einem konzeptuellen Objekt und einer Hierarchie von Implementierungsobjekten. Für jede einzelne Klasse, in deren Extent das Objekt liegt, ist eine eigene Instanz in der Menge enthalten, die die lokalen Methoden und Instanzvariablen dieser Klasse zur Verfügung stellt. Dies gilt auch für die Vererbungshierarchie; es wird also auch für jede Sub- bzw. Superklasse eine neue Instanz angelegt. Ein spezielles Dispatching-Verfahren übernimmt die Methodenausführung: Wird für ein bestimmtes Implementierungsobjekt eine gewünschte Methode nicht gefunden, so wird das konzeptuelle Objekt gesucht und aus der Menge aller Implementierungsobjekte dasjenige herausgesucht, das die gewünschte Operation anbietet.

Vergleich: Obwohl MultiView ein objekterhaltender Ansatz ist, stellt er nur wenig Funktionalität zur Verfügung, da immer nur minimale Änderungen an den Objekten möglich sind. Des weiteren handelt es sich um ein geschlossenes System. Alle Änderungen erfolgen innerhalb der Klassenhierarchie der Datenbank, es gibt keine Verweise auf externe Objekte, und die Unifikation von Klassen wird überhaupt nicht untersucht. Die Anforderungen an das darunter liegende Datenbanksystem sind sehr hoch und werden von keiner kommerziellen Datenbank zur Verfügung gestellt. Besonders auffällig sind dabei zwei negative Eigenschaften. Die erste ist die inflationäre Erzeugung neuer Klassen und Instanzen, da für jede Basisoperation mindestens eine neue Klasse erzeugt werden muß, die für das Object-Slicing vollständig mit neuen Objekten gefüllt wird. Die zweite ist der Mangel an Datenbankautonomie: Jede Sichtdefinition erfordert umfangreiche Änderungen an den bereits existierenden Datenbankklassen; Instanzvariablen werden zwischen Klassen migriert, um den Single Point of Inheritance zu gewährleisten, und sämtliche Instanzen dieser Klassen müssen entsprechend angepaßt werden. Die Verwaltung der Multi-Instanz-Objekte, das dazugehörige Dispatching sowie der Zugriff auf konzeptuelle Objekte bei Objektvergleichen muß ebenfalls in das Datenbanksystem integriert werden.

7.9 Objektsichten von Bertino

Elisa Bertino von der Universität Genf hat auf der EDBT’92 ebenfalls ein objektorientiertes Sichtsystem vorgestellt ([Ber92]). Eine Sicht ist dabei stets eine neue Klasse, die mit Hilfe einer

Anfrage aus anderen Klassen hergeleitet wird. Die neue Klasse wird dabei nicht, wie etwa bei COCOON, in den vorhandenen Typverband integriert, sondern stets als unabhängige Klasse dem Verband hinzugefügt — allerdings besteht die Möglichkeit, eine Liste von bereits existierenden Sichtklassen als Superklassen anzugeben, wodurch sich eine Sichthierarchie aufbauen läßt.

Eine Sichtdefinition hat das folgende Format:

create-view Viewname
[**properties** ListOfPropertyNames]
[**view-query** Query]
[**additionalproperties** ListOfProperties]
[**methods** ListOfMethodSpecifications]
[**superviews** ListOfViews]
[**OID** (**true** | **false**)]
[**identityfrom** ListOfClassNames]

Viewname ist der Name der neuen Sichtklasse, ListOfViews die Liste ihrer Superklassen, und ListOfPropertyNames listet alle Attributnamen und eventuelle Defaultwerte auf. Die Anfrage Query füllt wie im relationalen Fall die Klasse mit Objekten und Werten (siehe unten). Mittels additionalproperties können neue, nicht abgeleitete Attribute definiert werden, die wie in MultiView augmentierende Sichten ermöglichen. Das Verhalten der neuen Klasse wird durch eine Liste von Methodenspezifikationen festgelegt, die Methoden neu definieren oder von anderen Klassen importieren können. Der Parameter OID legt fest, ob die Sichtobjekte Oids bekommen sollen oder als strukturierte Werte aufgefaßt werden. Bei der Vergabe neuer Oids kann eine Liste von Klassennamen angegeben werden, die festlegt, von welchen Basisobjekten eines Joins die Objektidentität abgeleitet werden soll. Die Zuordnung zwischen Sichtobjekten und Basisobjekten erfolgt durch eine Systemfunktion namens baseobject, die jedoch nicht näher erläutert ist.[41] Die für das System verwendete Anfragesprache basiert auf einem objektorientierten Prädikatenkalkül (*Object-Oriented Predicate Calculus, OOPC*), stellt aber im Prinzip ein relationales Select-From-Where zur Verfügung.

Beispiel 7.3:

Die folgende Beispielanfrage liefert die Nachnamen aller festen und beratenden Mitarbeiter, die in einem Team arbeiten und ein monatliches bzw. tägliches Mindesteinkommen haben. Der letzte Ausdruck ist dabei ein sogenanntes *Alternativprädikat*, das bedingte Abhängigkeiten überprüft:

target *x*.name.l-name;
range *x* /(Permanent ∨ Consultant)
qualification (∃*t* /Team) (*x* **ISIN** *t*.tech-staff
∧ **CLASS_OF**(*x*) = [Permanent: *x*.monthly-wage>4000; Consultant: *x*.daily-wage>500]). ■

Die Schlüsselwörter target, range und qualification werden allerdings üblicherweise weggelassen und die drei Terme nur durch Semikola getrennt. Relationships können über das Schlüsselwort connection erzeugt werden, wobei eine existierende Relationship in die Sicht übernommen wird. Durch connection by base-instance werden Identitäts-Relationships zwischen zwei Sichtklassen erzeugt, die von derselben Basisklasse abgeleitet sind. Eine weitere Besonderheit ist die Unterscheidung zwischen direkten und indirekten Instanzen (*instances* bzw. *members*): Obwohl nicht zwischen Typ- und Klassenverband unterschieden wird, die Klassen also in der üblichen Inklusionshierarchie angeordnet sind, wird beim Zugriff auf einen Klassennamen standardmäßig nur

41. Hierzu wird auf einen technischen Bericht verwiesen [Ber91].

auf die direkten Instanzen zugegriffen; will man auch auf die Instanzen der Subklassen zugreifen, so muß dies durch die Verwendung eines Sternchens nach dem Klassennamen explizit angegeben werden. Eine vollständige Sichtdefinition sieht also zum Beispiel folgendermaßen aus:

create-view NewEmployee
properties (name, ssn, city)
view-query *x*.name, *x*.ssn, *x*.address.city; *x*/Employee*; *x*.age>40
additionalproperties (complete-address: NewAddress **connection** address)
methods (wage() → numeric, employer() **as** company() **of** Employee)
superviews NewPerson
OID true
identityfrom Employee.

In [Ber92] wird das Sichtsystem vorwiegend zur Schema-Evolution verwendet und bietet daher keinen Zugriff auf externe Objekte. Durch Kombination mit einer vordefinierten Methodenschnittstelle ([BNPS89]), die Objekt-Iteratoren und Zugriffsfunktionen definiert, wird es in [BNPS94] zu einer Integrationskomponente erweitert. Eine externe Klasse wird hierbei als abstrakte Klasse importiert und kann mit anderen abstrakten Klassen vereinigt werden. Der Zugriff auf die externen Klassen wird automatisch über die vordefinierten Funktionen erledigt, die Unifikation von Objekten erfolgt durch Re-Implementierung einer Methode namens *AreSame*, die berechnet, ob zwei Objekte von unterschiedlichen Datenbanken dasselbe Realweltobjekt darstellen.

Vergleich: Von der Grundidee ist der Ansatz von Bertino sehr ähnlich zu meinem eigenen. Objekte werden mit Hilfe einer Anfrage ausgewählt und umstrukturiert, mit einer neuen Oid versehen und in einen separaten Typverband eingeordnet. Zur Unterstützung von Updates wird eine Zuordnung zwischen Sichtobjekten und Basisobjekten verwaltet. Es gibt jedoch keinerlei Hinweise, wie diese Zuordnung vorgenommen wird, und wie mit Identitätserhaltung, mehrfacher Objekterzeugung und Vererbung umgegangen wird. Insgesamt ist die Umsetzung recht unklar, da für jede spezifische Eigenschaft irgendein neues Schlüsselwort eingeführt wird, das diese Eigenschaft irgendwie zur Verfügung stellt (siehe zum Beispiel connection). Man erfährt weder, was eigentlich genau passiert, noch hat man die Möglichkeit, in dieses Vorgehen einzugreifen. Die Verwendung einer einzelnen Anfrage für die Instanzmenge und alle Attributwerte führt zu einer komplexen Spezifikation und spricht für einen kopierenden und nicht navigatorischen Ansatz. Insgesamt ist die Sichtspezifikation durch die vielen Schlüsselwörter, langen Parameterlisten und die teilweise mathematische Formulierung sehr unübersichtlich. Darüber hinaus gibt es kein explizites Schema-Konzept, obwohl die erzeugten Sichtklassen Beziehungen untereinander aufbauen können.

7.10 VODAK (Klas et al.)

In Abschnitt 6.6 bin ich bereits auf das objektorientierte Datenbanksystem VODAK ([Kla93], [KFM*96]) des Fraunhofer IPSI eingegangen. VODAK selbst bietet kein eigenes Sichtsystem an, stellt aber erweiterbare Metaklassen zur Verfügung, mit denen weitere Klassenfunktionalität hinzugefügt werden kann. Die wesentliche Eigenschaft ist dabei das sogenannte *Message Passing*, das es erlaubt, Methoden abzufangen und an andere Klassen und Objekte zur Ausführung zu delegieren. In [KDN90] wurden unter dem Namen *ViewSystem* eine Reihe von Metaklassen zur Spezialisierung, Generalisierung, Gruppierung und Aggregation definiert, um einzelne Sichtoperationen verfügbar zu machen. Eine andere Veröffentlichung verwendet direkt das Message Passing, um mit Hilfe von Wissensbanken sinnvolle Interpretationen von Benutzeranfragen zu finden, die nur indirekt durch Pfadverfolgung im Schema auswertbar sind ([NeS88]). In

[KFA94] werden weitere Metaklassen definiert, um auf externe Sybase- und Postgres-Datenbanken zuzugreifen und so eine Datenintegration zu unterstützen. Ein vollständiges Sichtsystem ist in VODAK nie realisiert worden. Das System diente nur als Testumgebung zur Entwicklung neuer Konzepte und wird nicht weiter entwickelt. Die Integrationsmethodologie von P. Fankhauser ist unter anderem ebenfalls in diesem Umfeld entstanden ([Fan97], [KFM*96], siehe Seite 32).

7.11 UniSQL/X und UniSQL/M von Kim et al.[42]

In [KiK95] wird das objektorientierte Sichtsystem UniSQL/X von UniSQL, Inc. vorgestellt. Eine Sicht ist dort eine virtuelle Klasse mit Attributen, Methoden, einer optionalen Liste von Superklassen und einer SQL-Anfrage, die den Extent und die Attributwerte berechnet:

```
Create View Senior (Name, SSN, TotalPayment:real) AS
Select Name, SSN, Salary+Commission From Employee Where Age>40;
```

Die neu erzeugten Klassen bilden dabei eine eigene Hierarchie und werden nicht in den Verband der Datenbankklassen eingefügt. Im allgemeinen wird die Sichthierarchie jedoch eher flach bleiben, da bei einer Subklassenbeziehung die Sichtklassen mit disjunkten Extents definiert werden müssen — es wird also nur der direkte und nicht der volle Extent berechnet, was bei allen Anfragen beachtet werden muß. Die Arbeit mit den Sichtklassen erfolgt vorwiegend wertorientiert. Soweit es möglich ist, wird nur mit Hilfe der SQL-Anfragen ausgewertet. Es werden grundsätzlich keine neuen Oids erzeugt. Wenn eine Identifikation notwendig ist, wie etwa bei Relationships[43], wird die Oid des ursprünglichen Datenbankobjekts verwendet. Durch diese automatische Oid-Zuordnung werden sowohl Abgeschlossenheit bei Relationships als auch einfache Updates unterstützt. Sichtobjekte, die durch einen Join berechnet werden, können somit keine eigene Identität besitzen.

Das in [KGK*95] vorgestellte UniSQL/M stellt eine Erweiterung von UniSQL/X auf Multidatenbanken dar, mit der sowohl relationale als auch objektorientierte Datenbanken integriert werden können. Der Import erfolgt dabei über sogenannte *Proxys*, die auf der externen Datenbank eine Anfrage ausführen und die Ergebnisse als Objekte zur Verfügung stellen. Um Updates zu ermöglichen, sind Proxyklassen stets 1-zu-1-Kopien der Originalklassen mit minimalen Veränderungen, wie etwa Umbenennungen oder Skalierungen. Wie bei UniSQL/X erfolgt die Auswertung vorwiegend wertorientiert; werden Oids benötigt, so können die externen Oids bzw. Primärschlüssel dafür verwendet werden:

```
CREATE VCLASS Employee_Proxy1 ON LDB LocalDB1
    (Name string, SSN integer, Salary real)
    OBJECT_ID(SSN)
    AS SELECT e_name, e_ssn, 1.75*e_sal FROM emp;
```

Mit Hilfe weiterer virtueller Klassen können die Proxys weiter modifiziert und schließlich vereinigt werden. Die Vereinigung ist stets disjunkt — es kann keine Unifikation durchgeführt werden.

```
CREATE VCLASS Employee
    (Name string, SSN integer, Salary real)
    AS SELECT Name, SSN, Salary FROM Employee_Proxy1,
        SELECT Name, SSN, Salary FROM Employee_Proxy2;
```

42. Auf die Arbeiten desselben Autors am objektorientierten Datenbanksystem Orion wird in Kapitel 7.12 eingegangen.

43. Diese heißen hier *genestete Attribute*.

Besonderes Augenmerk richten die Autoren auf das Verhalten dieser Sichtklassen unter Schemamodifikationen.

Vergleich: Die Proxyklassen und virtuellen Klassen entsprechen meinen Import- und Konvertierungsklassen; die Trennung der Typhierarchien erfolgt ebenfalls wie in meinem Ansatz. Die große Einschränkung ist jedoch, daß das Objektverhalten zum großen Teil nur simuliert wird. Es stehen keine wirklichen, neuen Objekte zur Verfügung, sondern in erster Linie Datentupel, die bedingt mit einer Identität versehen werden können. In welcher Form Methoden, die hier als externe C-Funktionen implementiert sind, auf die Daten zugreifen können, wird nicht beschrieben.

7.12 Weitere interessante Arbeiten im Überblick

Ein weiteres ODMG-basiertes Sichtsystem wurde von *Elke Radeke* (Cadlab, Paderborn) auf der DEXA'96 vorgestellt ([Rad96]). Die Definition abgeleiteter Klassen basiert dabei nicht auf Anfragen, sondern erfolgt mit Schema-Operatoren und Methoden. Eine Sicht ist hierbei ein Schema, das seine Komponenten von anderen Schemas erbt. Durch die Kommandos drop und rename können einzelne Elemente ausgeblendet und umbenannt werden. Um strukturelle Änderungen vorzunehmen, müssen explizite C++-Funktionen programmiert werden, die die Konvertierung vornehmen:

```
schema MyView: DesignSchema, SalesSchema          // schema inheritance
{
drop Depot;                                       // remove class
drop Product::last_change;                        // remove attribute
rename DesignSchema::Product::kind name;          // rename attribute
Product::no = String2Int(SalesSchema::Product::no);  // define new attribute
int String2Int(String no) { /* C++ code... */ };  // conversion function
}
```

Dieser Ansatz hat den Nachteil, daß er keine deklarative Spezifikation und Auswertung erlaubt. Alle Umformungen müssen in einer Programmiersprache vorgenommen werden, so daß es keine Möglichkeiten zur Anfrageoptimierung gibt. Es gibt keine Extent-Spezifikationen; alle Sichtklassen sind 1-zu-1-Kopien der Originalklassen, ohne Selektions- oder Kombinationsmöglichkeiten. Es wird weder darauf eingegangen, wie die Objekte erzeugt und verwaltet werden, noch wie Objektidentität behandelt wird. Verwiesen wird hierzu auf das System Efendi, das entwikkelt wurde, um Daten zwischen heterogenen Datenbanken zu migrieren und transparent darauf zuzugreifen ([RaS95]).

In [CBR94] stellen *Armin Cremers et al.* ein Sichtsystem der Universität Bonn vor. Genau wie der vorige Ansatz wird auch hier auf die Angabe von Anfragen verzichtet und die Sicht mit Hilfe von Vererbung definiert, und zwar indem die Programmiersprache C++ um geeignete Konstrukte erweitert wird, die durch einen Präprozessor in reines C++ übersetzt werden. Genau wie bei COCOON steht die Update-Unterstützung und somit die Objekterhaltung im Vordergrund. Leider gehen die Autoren in dem Bericht nicht auf die Umsetzung des Systems ein, dessen Forderung nach mehrfacher Klassenzugehörigkeit im Widerspruch zu C++ steht.

In [ShS89] entwerfen *Shilling* und *Sweeney* vom Georgia Institute of Technology und IBM ein Sichtsystem speziell für Software-Entwurfs- und Wartungssysteme. Sie erweitern dafür das übliche Objektmodell um drei Konzepte, die aus jedem Objekt eine Art „Multiversions-Objekt" machen. Erstens definiert jede Klasse eine beliebige Anzahl von *Interfaces*, die jede ihrer Instanzen anbieten kann. Zweitens erfolgt eine Zuordnung aller Instanzvariablen (Datenfelder) auf die Menge der Interfaces, in denen sie verfügbar sind (*labelling*). Unterschiedliche Interfaces des-

selben Objekts können dadurch sowohl gemeinsame als auch individuelle Datenfelder enthalten, und sogar unterschiedliche Datenfelder mit denselben Namen. Drittens kann jedes Objekt eine beliebige Anzahl von *Kopien seiner Instanzvariablen* besitzen, die teilweise auch gemeinsam genutzt werden können. Eine *Sicht* ist dann eine konsistente Projektion einer Klassen-Teilmenge, in der jeder Instanz genau ein Interface und eine Kopie der Instanzvariablen zugewiesen wird. Durch die Definition mehrerer Sichten kann auf dasselbe Objekt mit unterschiedlichen Schnittstellen und (teilweise) unterschiedlichen Daten zugegriffen werden. Eine Prototypimplementierung war auf C++ geplant, indem jedes Multiversions-Objekt auf eine komplexe C++-Objektstruktur aus einem *Master Object*, einer Reihe von *Slave Objects* und einem Verwaltungsmodul für Instanzvariablen abgebildet wird. Dieser Ansatz stellt somit hohe Anforderungen an das Objektmodell und ist dadurch nicht allgemein einsetzbar.

In [SaH96] stellen *Stefan Sander* und *Willi Hasselbring* von der Universität Dortmund ein geplantes föderiertes Datenbanksystem auf Basis von ODMG–93 und CORBA vor. Genau wie in IRO-DB soll ODMG–93 als kanonisches Datenmodell zur Datenbankintegration dienen, die Netzzugriffe sollen dabei jedoch mit CORBA erfolgen. Die Autoren bemängeln das Fehlen eines Sichtsystems in ODMG, stellen jedoch keine konkrete Lösung dieses Problems vor. Im Gegensatz zu meinem System fordern sie, daß Oids lebenslange Gültigkeit haben, wobei die Konsistenz globaler Oids mit Hilfe von Triggern realisiert werden soll. Das Projekt scheint nie umgesetzt worden zu sein; zumindest sind keine weiteren Veröffentlichungen zu finden.

Ein interessanter Ansatz zur Behandlung von Objektidentität bei der Unifikation von Objekten stammt von *Qiming Chen* und *Ming-Chien Shan* von den HP Laboratories ([ChS93]).[44] Ein Objekt wird dort nicht mehr durch eine einzelne Oid identifiziert sondern durch eine Menge von Oids. Zwei Objekte aus unterschiedlichen Quellen können integriert werden, indem man die dazugehörigen Oid-Mengen vereinigt. Das integrierte Objekt „erbt“ dabei alle Funktionen der Originalobjekte und kann zusätzlich eigene Funktionen definieren. Die Enthaltenseinsbeziehung der Oid-Mengen impliziert dabei die Vererbungshierarchie und somit einen Typverband. Obwohl der Ansatz eine sehr elegante Lösung für die Zuordnung zwischen Sicht- und Originalobjekten liefert, ist er nur sehr eingeschränkt einsetzbar. Er erlaubt nur eine 1-zu-1-Unifikation verschiedener Quellen; man kann weder spezifische Zuordnungen definieren noch separate Objektteile extrahieren.

In [ByM93] stellen *Kwang June Byeon* und *Dennis McLeod* von der University of Southern California, Los Angeles ihre sogenannten *virtuellen Datenbanken* vor. Sie argumentieren, daß Sichten, im Sinne einer Schemaanpassung an externe Anwendungen, und Versionierung, als zeitlich bedingte Weiterentwicklung der Datenbank, auf denselben Prinzipien beruhen und unter dem Oberbegriff der virtuellen Datenbank zusammengefaßt werden sollten. Abgesehen von der Wahl der Begriffe deckt sich dies mit der auf Seite 18 beschriebenen Anwendung von Sichten zur Versionierung. Das vorgestellte System basiert auf dem *Core Object Data Model CODM* mit Objekten in einer einfachen Klassenhierarchie. Ein virtuelles Schema ist ein in sich abgeschlossenes Schema, das aus importierten, abgeleiteten und lokalen Klassen besteht. Es wird interaktiv in fünf Schritten erzeugt: 1) Import der Klassendefinition, 2) Überprüfung und Ergänzung der transitiven Hülle aller Datenfelder, 3) Überprüfung und Vervollständigung der Vererbungshierarchie, 4) Import der Instanzen, 5) Transformation der Instanzen. Abgesehen von einem Hinweis auf ein *Remote Exchange Experimental System* ([FGMS93]) als Basis für die Implementierung wird nicht auf die Umsetzung auf Instanzebene eingegangen. Einerseits sprechen die Autoren von einem reinen Kopiervorgang der Daten, andererseits sollen auch die Basisobjekte verändert

44. Siehe auch Pegasus, S. 144.

werden können, sofern die virtuelle Datenbank mit sogenannter „starker Abhängigkeit“ definiert worden ist. Es wird weder auf Objektidentität noch auf die notwendige Migrationsfähigkeit der Datenbankobjekte eingegangen. Die Grundideen sind somit vergleichbar mit meinem Ansatz, das Gesamtsystem ist mit der interaktiven Schemamanipulation aber eher rudimentär und unvollständig.

In [MaM91] ergänzen *Jean-Claude Mamou* und *Claudia Bauzer Medeiros* vom Altaïr-Konsortium objektorientierte Sichtdefinitionen um eine Präsentationskomponente und nennen die Kombination dann *Hyper-View*. Der Prototyp *ToonMaker* analysiert eine solche Sichtdefinition und erzeugt automatisch eine geeignete Bildschirmdarstellung für die Daten, die sowohl die reine Darstellung als auch Eingabefelder für Datenmanipulationen einschließt. Das System ist eine Erweiterung von O_2 und eher als Arbeitshilfe für interaktive Datenbankzugriffe auf strukturierten Daten gedacht. Ein objektorientiertes Sichtkonzept wird nicht vorgestellt.

Ein sehr eingeschränktes Sichtmodell wird in [YoC93] von *Suk I. Yoo* und *Hai Jin Chang* von der Seoul National University vorgestellt. Sie definieren eine Sicht nicht als vordefinierte Klasse oder ein Schema sondern, wie im klassischen relationalen Fall, als eine benannte Anfrage. Die verwendete Anfragesprache unterstützt Selektion, Projektion und Kombination von Objekten, erzeugt aber keine neuen Klassen. Durch ein spezielles Dispatching-Verfahren, bei dem alle Methodenausführungen über eine spezielle Funktion RunMethod ablaufen, der die (generische) Instanz, der Methodenname und die Parameter übergeben werden, können temporär auch neue Methoden definiert werden, indem sie als benannte Lambda-Funktionen in der Anfrage definiert und an die Objekttupel angehängt werden. Anfragen an eine solche Sicht können dann vollständig mit Hilfe von Anfrageexpansion ausgewertet werden. Die so erzeugten Sichten sind jedoch sehr unübersichtlich, und es gibt weder Vererbung noch Konsistenzbedingungen. Die Verwendung von Lambda-Funktionen erfordert ein funktionales Datenmodell und führt durch den expliziten Aufruf mittels RunMethod leicht zu Laufzeitfehlern.

Eine der ersten Veröffentlichungen über objektorientierte Anfragesprachen stammt von *Banerjee*, *Kim* und *Kim* ([BKK88]). Sie definiert zwar keine Sichtfunktionalität, diskutiert aber bereits viele Eigenschaften eines solchen Systems auf Basis der objektorientierten Datenbank *Orion* von MCC. Das einfache Datenmodell geht von einer Einheit von Typ und Klasse aus, trennt aber deutlich zwischen direkten und indirekten Instanzen: Man muß bei der Anfrageauswertung angeben, ob man nur die direkten Instanzen einer Klasse betrachten möchte, oder auch alle Instanzen ihrer Subklassen — wobei das Ergebnis dann ebenfalls aus einer Hierarchie von Objekten besteht. Die Anfragesprache definiert in erster Linie eine reine Selektionsfunktionalität auf einzelnen Klassen und der transitiven Hülle ihrer Attribute.[45] Die Projektion wird zwar erwähnt, es wird aber nicht klar, wie Attribute aus den Ergebnisobjekten entfernt werden sollen, bzw. ob die Auswertung rein wertorientiert erfolgt. Auf Methoden wird nicht eingegangen. In [Kim89] kommen zur reinen Navigation noch zwei Join-Möglichkeiten hinzu: Zum einen wird die Existenz einer impliziten Inversen für jedes Attribut postuliert, um Relationships auch in umgekehrter Richtung durchlaufen zu können, zum anderen wird ein expliziter Join definiert, der als Ergebnis eine Menge von Oid-Tupeln zurückliefert.[46] Im Gegensatz zu reinen Anfragen, die die Oids existierender Objekte zurückliefern, müssen bei der Erzeugung eines Snapshots neue Objekte erzeugt werden. Hierzu wird jeweils eine neue Klasse (oder eine Klassenhierarchie mit allen Subklassen) erzeugt, die nicht in den existierenden Klassenverband eingebettet wird. Man gelangt

45. Interessant ist hierbei der recurse-Operator, der die rekursive Hülle eines reflexiven Attributs bestimmt.

46. Kim spricht von *heterogenen Listen*. Das Ergebnis eines Joins ist eine flache Oid-Liste, die keine Gruppierung enthält. Die Anwendung muß selbst entscheiden, wo ein Oid-Tupel aufhört und das nächste anfängt.

hier also in den objektgenerierenden Fall, wobei jede Verbindung zu den Originalobjekten verloren geht. Unabhängig von dieser Anfragesprache wird in [BKKK87] eine vollständige Taxonomie möglicher Schemaevolutionsoperationen angegeben und für Orion umgesetzt. Durch die Verwaltung von Kontextinformationen und zusätzliche Indirektionen beim Attributzugriff können mehrere Versionen der Objekte gehalten werden. Auf eine Kombination von Evolution und Anfragesprache wird nicht eingegangen, obwohl sie zu objekterhaltenden Sichten führen könnte.

In [KiT91] stellen *Yutaka Kimura* und *Kunitoshi Tsuruoka* vom NEC C&C Systems Research Laboratory einen rein prozeduralen Sichtansatz für ihr objektorientiertes Datenbanksystem *Odin* vor. Odin ist eine Erweiterung von C++ mit persistenten und abgeleiteten Klassen. Grundlage des Systems sind sogenannte *Mengenobjekte (set objects)*: Für jede Klasse C kann mittels $C das zugehörige Mengenobjekt, also der Extent bestimmt werden. Eine Sicht ist eine spezielle C++-Klasse, bei der dieser Extent mit einer Meta-Funktion namens derive() berechnet wird. Das folgende Beispiel definiert eine Selektions-Sicht, die alle Personen auswählt, die mindestens 65 Jahre alt sind, und alle Attribute bis auf Name und Adresse ausblendet:

```
view class Senior { :public meta void derive(); }
meta void Senior::derive()
   { $Person.select( [self:S; | view S(name;address); where S.age>=65;] );}
```

select() liefert die Originalinstanzen der Basisklasse und ist somit objekterhaltend. Das neue Verhalten wird durch ein spezielles Dispatching-Verfahren der Datenbank erreicht: Jeder Methodenaufruf wird zur Basisklasse des Objekts delegiert. Diese Basisklasse verwaltet eine Liste der auf ihr definierten Sichten und wählt mit Hilfe von Transaktionsinformationen eine konkrete Methode aus. Von der Basisklasse übernommene Methoden werden *primitive Methoden* genannt, zusätzlich können weitere Methoden definiert werden (*additional methods*); ein Hinzufügen von Attributen ist nicht möglich. Neben der Selektion gibt es auch noch eine Join- und eine Group-Methode, die jeweils sogenannte *Tupelobjekte* zurückliefern. In allen Fällen ist die Sichtklasse eine direkte Subklasse der allgemeinsten Datenbankklasse Object und wird nicht weiter in den Klassenverband integriert. Insgesamt sind die Sichtmöglichkeiten sehr eingeschränkt. Es gibt weder deklarativen Zugriff noch werden Integration oder strukturelle Umformungen unterstützt. Das spezielle Dispatchingverfahren erfordert darüber hinaus einen Eingriff in das Datenbanksystem.

Bernhard Schiefer vom FZI Karlsruhe beschreibt in [Sch93] ein Sichtsystem auf Basis des objektorientierten Datenbanksystems OBST ([CRS*92]). Es ähnelt dem Ansatz von COCOON, da es strikt objekterhaltend ist und eine automatische Reklassifikation auf Basis von Klassenprädikaten verwendet. Im Gegensatz zu COCOON werden die Sichtklassen jedoch nicht in den vorhandenen Klassenverband eingebettet sondern bilden jeweils einen eigenen Verband. Da es durch die mehrfache Klassenzugehörigkeit nun zu Mehrdeutigkeiten bei der Methodenausführung kommen kann, muß in Zweifelsfällen eine künstliche Klasse eingefügt werden, die ein explizites Methoden-Dispatching durchführt. Ein weiteres Manko ist, daß ausschließlich objekterhaltende Sichten erlaubt sind, so daß Joins und größere Umstrukturierungen nicht möglich sind; außerdem unterstützt OBST keine deklarative Anfragesprache. In [Sch93] wird gezeigt, daß mit Hilfe dieser Sichten Schemaevolution unterstützt werden kann, indem das alte Schema jeweils als Sicht auf dem veränderten Schema für alte Anwendungen zur Verfügung gestellt wird (siehe Seite 18). Der dort ebenfalls verwendete Begriff der Integration ist hingegen irreführend, da es sich nicht um Datenbankintegration handelt — die mit dem vorgestellten System aufgrund seiner strikten Objekterhaltung überhaupt nicht durchführbar ist — sondern um die Einbettung neuer Anwendungen in das System, wobei die erforderlichen Datenstrukturen in Form von Sichten hinzugefügt werden.

Zhiyong Peng und *Yahiko Kambayashi* von der Universität Kyoto stellen in [PeK95] sogenannte *Stellvertreterobjekte* (*deputy objects*) vor, mit denen sie sowohl Sichten als auch Rollen und Schemaevolution realisieren. In diesem theoretischen Ansatz werden wie bei meinem Ansatz neue Objekte erzeugt, die strukturelle Änderungen an den Originalobjekten darstellen. Für jedes Originalobjekt wird vom System ein vollständiger Baum all seiner Stellvertreterobjekte verwaltet und dynamisch modifiziert. Die Zuordnung zwischen den Objekten erfolgt durch vollständige Spezifikation sämtlicher Abbildungsfunktionen und ihrer Inversen. Einerseits können hierdurch Updates in alle Richtungen weitergeleitet werden, andererseits erfordert dieses Verfahren einen extrem hohen Verwaltungsaufwand und muß bereits von der Originaldatenbank bereitgestellt werden; es ist folglich nicht möglich, ein solches Sichtsystem auf einer autonomen Datenbank aufzusetzen. Weiterhin können keine Joins sondern ausschließlich 1-zu-1-Ableitungen dargestellt werden, und ein Zugriff auf externe Datenbanken ist ebenfalls nicht vorgesehen.

Das von *Tanaka*, *Yoshikawa* und *Ishihara* in [TYI88] vorgestellte Sichtsystem nutzt das Metaschema und die Typlosigkeit von Smalltalk für reorganisierte Selektionssichten aus. Eine *virtuelle Klasse* wird mit Hilfe eines Selektionsprädikats definiert und repräsentiert eine Menge von Originalobjekten. Durch explizite Angabe von ISA-Beziehungen können die virtuellen Klassen in eine Hierarchie eingebettet werden; ein sogenanntes *virtuelles Schema*. Die hierbei geforderten Konsistenzregeln sind allerdings durch ihre prädikatenlogische Vollständigkeit nur in Ausnahmefällen wirklich überprüfbar. Die streng hierarchische Klassenzugehörigkeit von Smalltalk wird dadurch umgangen, daß virtuelle Klassen nicht als Smalltalk-Klassen sondern als eine Art Container-Objekte definiert werden. Alle Instanzen bleiben in ihren Originalklassen, und nur wenn eine Methode nicht gefunden wird, wird die Fehlermeldung abgefangen und in den virtuellen Klassenobjekten nach einer Implementierung gesucht. Dieser Ansatz widerspricht somit dem eigentlich gewünschten Sichtverhalten. Der umgekehrte Ansatz, mit den virtuellen Klassen zu beginnen und erst danach auf die Originaleigenschaften zuzugreifen wird nur als mögliche weitere Entwicklung angesprochen. Eine weitere Einschränkung ist, daß jedes Objekt eine Liste seiner Sichtklassen verwalten muß.

In [Day89] stellt *Umeshwar Dayal* zwei Anfragesprachen für das funktionale OODAPLEX vor und geht dabei kurz auf Sichten ein. Er unterscheidet zwischen *unabhängigen* Sichten, die nur einmal berechnet werden und dann jeglichen Bezug zu den Originalobjekten verlieren, und *abhängigen* Sichten, die eng mit ihren Originalobjekten gekoppelt sind. In bestimmten Situationen sollen dabei die Sichtobjekte berechnete Oids erhalten, die in Form von Tupeln die Originalobjekte enthalten, also meinem Orig-Attribut entsprechen. Weiterführende Veröffentlichungen zu dieser kurzen Anmerkung gibt es nicht.

Schemaevolution:

Eine sehr fundierte Untersuchung zur Anwendung von Sichten zur Schemaevolution stammt von *Zohra Bellahsene* vom LIRMM an der Universität von Montpellier ([Bel96]). Eine solche Anwendung fordert sowohl Identitätserhaltung als auch Kapazitätserhöhung.[47] Durch spezielle Import- und Export-Möglichkeiten werden Sichten einer Datenbank zu einer Föderation zusammengefaßt, um Redundanzen in unterschiedlichen Sichten zu verringern. Eine Übersicht über die Auswirkungen einzelner Schemamodifikationen und ein Vergleich mit anderen Ansätzen runden den Beitrag ab. In [Bel97] wird das System von einer einzelnen Datenbank auf eine Datenbankföderation übertragen. In einer neueren Arbeit untersucht die Autorin die Möglichkeit von Updates auf objektgenerierenden Sichten ([Bel00]). Sie unterteilt die dort auftretenden Joins

47. *Capacity Augmentation*; das Hinzufügen neuer persistenter Attribute zu Sichtklassen.

in schwache *Assoziationen* und starke *Aggregationen* und nutzt diese Information für eine unterschiedliche Inversenbildung beim Update. Dieser Ansatz kann möglicherweise verwendet werden, um mein System um deklarative Updates zu erweitern. Eine gute Übersicht über allgemeine Probleme und Ansätze zur objektorientierten Schemaevolution, allerdings ohne die Anwendung von Sichten, wird von *Nguyen* und *Rieu* in [NgR89] gegeben. Ein Panel zu diesem Thema ist im SIGMOD RECORD zu finden ([JTTW88]). Details zur Konsistenzerhaltung in einem solchen Umfeld findet man in [DeZ91] und [ABDS94].

Datenbankintegration:

Die Veröffentlichungen zur Datenbankintegration befassen sich in erster Linie mit den methodischen Problemen der Erkennung von Gemeinsamkeiten und der Konfliktauflösung. Eine Übersicht über die unterschiedlichen Schemaintegrationsmethodologien findet man zum Beispiel in [BLN86], eine Übersicht über die Verwendung von Sichten in diesem Zusammenhang in [Hul97]. Die Anwendung der Objektsichten von Bertino zur Integration sind bereits vorn beschrieben worden.

Xuequn Wu vom Fraunhofer ISST verwendet für die Datenbankintegration eine modellbasierte Abbildung der Datenbanken ([Wu96a], [Wu96b]). Für jede externe Datenbank werden die Relationen und Tupel im Objektmodell dargestellt und nicht die dahinterliegenden Konzepte, wie etwa Personen oder Adressen. In den bisherigen Veröffentlichungen erfolgt neben der reinen Bereitstellung dieser Objekte keine weitere Integration. Die *Multidatabase View Definition Language (MVDL)* von *Duwairi et al.* von der University of Wales ist eher formaler Natur ([DFG96], [Duw97], [AGF03]). Sie definiert Basisoperatoren zur Integration von Klassen, wie etwa Vereinigung, Durchschnitt oder Join, und fügt zusätzliche Korrespondenzaussagen in Form von Prolog-Regeln hinzu. Das System ist eher interaktiv angelegt; eine deklarative Schemadefinition wird nicht beschrieben. Zum Umgang mit Objektidentität und Verteilung gibt es keine Aussagen.

Ebenfalls regelbasiert ist das *Multi-User View Integration System MUVIS* von *Hayne* und *Ram* von der University of Arizona, Tucson ([HaR90]). Sie verwenden eine Wissensbank mit Regeln und Synonyminformationen, um Ähnlichkeitsmaße zu berechnen und hieraus eine Integration abzuleiten. Das Integrationsverfahren basiert auf dem Ansatz von *Navathe et al.* ([NEL86]), das Datenmodell ist ein erweitertes ER-Modell mit einer Abbildung auf das semantische Datenmodell *SDM* von *Hammer* und *McLeod* ([HaM81]). Auf die konkrete Sichtdefinition und ihre Eigenschaften wird leider nicht eingegangen. Ein neuerer, rein formaler Ansatz zur automatischen Integration auf Basis von Korrespondenzaussagen stammt von *Yangjun Chen* und *Wolfgang Benn* ([ChB99]).

7.13 Zusammenfassung

Einen Vergleich der jeweiligen Ansätze mit meinem eigenen Sichtsystem habe ich bereits in den einzelnen Abschnitten gegeben. Die folgenden Tabellen fassen noch einmal die wichtigsten Kriterien im Sinne der Kategorisierung von Abschnitt 2.2.3 zusammen. Tabelle 6 beschreibt die Eigenschaften der vorgestellten Sichtsysteme:

- Die Art der Ergebnisses (wertbasiert, objekterhaltend, objektgenerierend)
- Die Granularität des Ergebnisses (Wert, Variable, Klasse, Schema)
- Die Einbettung in den vorhandenen Typverband (ja, nein)

- Die Unterstützung von Datenbankintegration durch Zugriff auf externe Objekte (überhaupt nicht, als Erweiterung, nur disjunkte Vereinigung, vollständig)

System	Ergebnistyp	Granularität	Einbettung	Integration
eXoT/C	objekterhaltend	Schema	nein	Erweiterung
FUGUE	objekterhaltend, objektgenerierend	Klasse	nein	nein
MIND	wertbasiert	Schema	nein	ja
COCOON	objekterhaltend	Klasse	ja	Erweiterung
OSCAR	objekterhaltend, objektgenerierend	Klasse	ja	nein
Abiteboul	objekterhaltend, objektgenerierend	Schema	nein	disjunkt
Pegasus	objekterhaltend, objektgenerierend	Schema	ja (in Sicht)	ja
Multiview	objekterhaltend, objektgenerierend	Schema	ja	nein
Bertino	objektgenerierend	Klasse	nein	nein
UniSQL/X&M	wertbasiert, objekterhaltend	Klasse	nein	disjunkt
Mein Ansatz	identitätserhaltend	Schema	nein	ja

Tabelle 6: Eigenschaften der Sichtsysteme

Tabelle 7 stellt dem die Anforderungen gegenüber, die die jeweiligen Systeme an das zugrunde liegende Datenmodell stellen:

- Typzugehörigkeit (einfach, hierarchisch, frei)
- Migrationsfähigkeit (keine, aufwärts, hierarchisch, frei)
- Erweiterbarkeit des Datenbankschemas (keine, eingeschränkt, frei)

Die letzte Spalte enthält jeweils noch einen kurzen Kommentar zu charakteristischen Merkmalen des jeweiligen Systems.

System	Typhierarchie	Migration	Erweiterbarkeit	Kommentare
eXoT/C	hierarchisch	keine	keine	eigenes Dispatching-Verfahren
FUGUE	frei	frei	keine	funktionales Datenmodell
MIND	keine	keine	keine	Objekte werden nicht instanziiert
COCOON	frei	frei	frei	nur 1-zu-1–Integration
OSCAR	frei	frei	frei	Clustermodell, Sichten nur am Rand
Abiteboul	frei	frei	keine	keine Vererbung zw. virt. Klassen
Pegasus	frei	frei	frei	reines Integrationssystem, Import relationaler Daten als Objekte
Multiview	frei	frei	frei	inflationäre Klassenerzeugung, eigenes Dispatching-Verfahren
Bertino	einfach	keine	keine	baseobject als Rückverweis
UniSQL/X&M	einfach	keine	keine	simuliertes Objektverhalten
Mein Ansatz	einfach	keine	keine	

Tabelle 7: Anforderungen der Sichtsysteme und Kommentare

Insgesamt läßt sich sagen, daß die meisten Ansätze die objekterhaltende Variante bevorzugen, gegebenenfalls ergänzt um objektgenerierende Operatoren für Gruppierungen. Dieser Ansatz stellt auf jeden Fall die eleganteste und konsistenteste Umsetzung unterschiedlicher Ausprägungen einzelner Objekte dar. Auf der anderen Seite muß er jedoch um weitere Mechanismen ergänzt werden, um auf externe Objekte zuzugreifen oder Objekte zu unifizieren. Außerdem stellen diese Ansätze hohe Anforderungen an das zugrunde liegende Datenbanksystem und sind somit bei strengen Sprachbindungen in statisch getypte Sprachen nicht einsetzbar. In fast allen Systemen wird auf ein eigenes Dispatching-Verfahren zurückgegriffen, um die Kontextwechsel zwischen Sicht und Datenbank zu realisieren. Hierdurch entsteht ein Bruch zwischen der Behandlung transienter Objekte der Programmiersprache und persistenter Objekte der Datenbank. Als Granularität treten Klassen und Schemas in gleicher Weise auf, wobei beide sowohl eingebettet als auch separat verwendet werden — vollständige Sichtschemas werden dabei vorwiegend separat gehalten und nicht in den Originalverband eingebettet.

Weitere Vergleiche objektorientierter Sichtmodelle findet man in der Dissertation von Michael Dobrovnik [Dob95] und in dem Artikel von Renate Motschnig-Pitrik [Mot96]. Neuere Arbeiten im Bereich der Datenintegration befassen sich so gut wie nicht mehr mit objektorientierten Datenbanken sondern vorwiegend mit frei strukturierten und flüchtigen Datenquellen. Hierauf gehe ich im Rahmen des Ausblicks ein.

8 Zusammenfassung und Ausblick

Die Aufgabe dieser Arbeit war die Analyse vorhandener Sichtansätze für objektorientierte Datenbanksysteme, der Entwurf eines Sichtsystems für die Integration statisch getypter Objektbanken, sowie die Umsetzung des Systems für die standardisierte Datenbankumgebung ODMG mit C++ als Programmiersprache.

Die in Kapitel 2 vorgenommene Analyse hat gezeigt, daß beim objektorientierten Datenmodell sehr viel mehr Aspekte bei der Erzeugung und Auswertung virtueller Elemente zu beachten sind als beim relationalen Modell. Aus diesen Aspekten habe ich eine Reihe von Eigenschaften objektorientierter Sichten definiert, mit denen man die unterschiedlichen Ansätze klassifizieren kann, was in Kapitel 7 geschehen ist. Der wichtigste Aspekt ist dabei der Umgang mit Objektidentität: Während *objekterhaltende* Ansätze zwar eine gute Modelliermöglichkeit für Sichtkonzepte bieten, fordern sie auf der anderen Seite umfangreiche Funktionalität vom verwendeten Datenbanksystem, um überhaupt anwendbar zu sein; hierzu gehören unter anderem freie Klassenzugehörigkeit, freie Migration, und im allgemeinen ein kontextabhängiges Dispatching-Verfahren, um verschiedenen Ausprägungen eines Objekts die richtigen Methoden zuzuordnen. Der ODMG-Standard bietet diese Funktionalität jedoch nicht an, und gleiches gilt auch für andere statisch getypte Objektbanken. Die *wertbasierten* und *objektgenerierenden* Ansätze sind wesentlich leichter umsetzbar, dafür verliert man die Zuordnung zwischen der Sicht und den ursprünglichen Datenbankobjekten, wodurch keine direkten Updates möglich sind.

Um nun objekterhaltendes Verhalten auf Basis eines technisch objektgenerierenden Systems zur Verfügung zu stellen, habe ich in Kapitel 3 den *verallgemeinerten identitätserhaltenden Ansatz* entworfen. *Identitätserhaltung* stellt dabei eine Verallgemeinerung von Objekterhaltung dar, indem innerhalb von virtuellen Klassen neue Objekte erzeugt werden, diese aber im Rahmen der Sicht die Identität der Originalobjekte übernehmen. Das System stellt dabei automatisch eine Zuordnung zwischen Sicht- und Originalobjekt her, was in meinem System durch *Rückverweise* erfolgt. Diese Zuordnung ist dabei sehr flexibel; neben einer einfachen eins-zu-eins-Zuordnung von Objekten können auch Objektmengen und Objektteile in der Sicht als eigenständige Objekte auftreten, wobei der Identitätsbegriff auf Äquivalenzklassen von Objekten ausgedehnt wird. Weiterhin erlaubt dieser Ansatz den Zugriff auf externe Objekte, was Datenaustausch und Datenbankintegration erst ermöglicht, ohne dabei die Autonomie der beteiligten Datenbanken zu verletzen.

Bei der Spezifikation kombiniere ich die Vorteile von deklarativen und prozeduralen Ausdrükken: Die deklarativen Herleitungen sind im allgemeinen leicht zu formulieren und erlauben durch Anfrageexpansion den Einsatz herkömmlicher Optimierer. Die prozedurale Spezifikation einzelner Methoden erhöht die Ausdrucksmächtigkeit für komplexere Herleitungen und erlaubt die Angabe konkreter Update-Verfahren, die durch automatische Inversenbildung nicht möglich wären. Die Definition vollständiger Sichtschemata mit zusätzlichen Integritätsbedingungen sichert Konsistenz und Abgeschlossenheit zu. Weitere wichtige Eigenschaften sind die Kompositionalität, also die Definition von Sichten auf Sichten, und die Unterstützung verschiedener Integrationsgrade.

Die in Kapitel 4 beschriebene Implementierung demonstriert die Anwendbarkeit des Ansatzes. Sie ist im Rahmen des ESPRIT-III-Projekts IRO-DB erfolgt und verwendet die wichtige C++-Sprachbindung des Datenbankstandards ODMG–93. Neben der vollständigen Spezifikation einer Objektverwaltung für die Identitätserhaltung und die Zusicherung referenzieller Integrität wurden auch spezifische Probleme beim dynamischen Objektzugriff gelöst. Durch die

Verwendung eines Präprozessors werden viele schemaspezifische Methoden automatisch erzeugt und der letztendliche Programmieraufwand minimiert.

Da meine Sichten grundsätzlich Updates unterstützen, habe ich in Kapitel 5 konkurrierende Updates untersucht und gezeigt, daß die vorhandenen Sperr- und Transaktionsmechanismen der beteiligten Datenbanken für die Konsistenzerhaltung ausreichen. Das vorgestellte System ist für ODMG–93 und C++ entworfen und implementiert worden, kann aber auch auf andere Umgebungen übertragen werden. In Kapitel 6 habe ich daher die notwendigen Änderungen untersucht, um das System an neuere Versionen von ODMG, an Java und an andere Datenbankformen anzupassen. Eine Umsetzung für CORBA ist erfolgreich durchgeführt und als Konferenzbeitrag veröffentlicht worden ([RBPF98]). Im abschließenden Vergleich mit verwandten Arbeiten zeige ich schließlich, daß keine der bisher existierenden Arbeiten die von mir angebotene Funktionalität einer flexiblen Datenbankintegration auf Basis einer üblichen, statisch getypten Programmierumgebung zur Verfügung stellt. Fast immer werden sehr hohe Anforderungen an das zugrunde liegende Datenbanksystem gestellt, die nur von den Prototypen der Entwickler geleistet werden können. In anderen Fällen verliert man die Update-Fähigkeit oder muß sogar das verwendete Datenmodell verlassen.

Mögliche Weiterentwicklungen meines Systems

Der wichtigste Aspekt für eine Weiterentwicklung des vorliegenden Sichtsystems ist die Untersuchung der Seiteneffekte von Methoden. Die notwendige Invalidierung externer Objekte bei jeder externen Methodenausführung stellt das größte Problem für einen effizienten Einsatz dar. Die möglichen Ansätze für eine Lösung des Problems — erweiterte Metainformationen oder dynamische Änderungslisten — sind bereits in Abschnitt 5.5 beschrieben worden.

Ein anderer Punkt ist die Erweiterung der deklarativen Spezifikationsmöglichkeiten. Im bisherigen System sind — teilweise aufgrund der Einschränkungen von ODMG — nur die Leseoperationen deklarativ spezifizierbar. Alle Updates sowie besonders komplexe Herleitungen müssen derzeit von Hand ausprogrammiert werden. Für eine mögliche Verbesserung sehe ich drei Ansätze:

1. Die Analyse der Herleitungsklauseln, um daraus automatisch oder halbautomatisch geeignete Update-Operationen zu generieren. Als Grundlage bietet sich die in Kapitel 7 vorgestellte Arbeit von Zohra Bellahsene zur Inversenbildung bei Assoziationen und Aggregrationen an.
2. Eine Erweiterung der grafischen Integrationskomponente, um aus den durchgeführten Integrationsschritten sowohl die Lese- als auch die Schreiboperationen vollständig herzuleiten.
3. Der Entwurf einer geeigneten Spezifikations-Metasprache, die speziell klassifizierte Operatoren für die unterschiedlichen Integrationsschritte anbietet, aus denen sowohl die Anfrage- und Update-Ausdrücke als auch der zugehörige C++-Code abgeleitet werden können. Ein Nebeneffekt einer solchen *Integrationssprache* ist auch die bessere Wartbarkeit und somit Evolutionsfähigkeit der Spezifikation, da nicht nur die technische Durchführung der Integration sondern zu großen Teilen auch das dahinter liegende Konzept (die Intention) erkennbar bleibt.

Allgemeine Entwicklung

ODMG ist der einzige objektorientierte Datenbankstandard geblieben und hat inzwischen einen relativ stabilen Status erreicht. In kommerzielle Produkte hat er aber noch keinen spürbaren Ein-

zug gefunden. Der in diesem Zusammenhang häufig verwendete Begriff der „ODMG-Compliance“ bedeutet fast immer nur, daß für die Datenbank eine Bibliothek angeboten wird, die eine annähernd ODMG-konforme Sprachbindung anbietet. In Bezug auf Datenmodell und Anfragesprache gibt es noch keine Vereinheitlichung.

Im Bereich der objektorientierten Datenintegration sind immer noch viele Fragen offen. Auf semantischer Seite mangelt es noch immer an Untersuchungen zur Integration von Methoden sowie an der automatischen Unterstützung bei der Integration von Objektnetzen mit Vererbungs- und Referenzbeziehungen. Auf technischer Seite fehlt noch immer ein sinnvoller Objektaustausch. Mit RPC, CORBA und OLE kann man zwar auf entfernte Objekte zugreifen, ein echter Import solcher Objekte über Rechner- oder Architekturgrenzen hinweg ist hiermit aber nicht möglich. Die zunehmende Verbreitung von Java mit seiner Objektserialisierung und Selbstbeschreibung (introspection) weist immerhin in diese Richtung. Ein Forschungsansatz, das von Manola et al. vorgeschlagene „RISC“ Objektmodell als größter gemeinsamer Nenner anderer Objektsysteme, stellt hingegen zu hohe Anforderungen an das Meta-Modell und dynamische Funktionen und Typen um realistisch umsetzbar zu sein ([MaH92], [MHG*92]).

Neuere Forschungsarbeiten

Die Forschungsschwerpunkte im Bereich der Datenintegration haben sich inzwischen verlagert. Vor allem durch die immense Datenflut mit der weltweiten Verbreitung des World Wide Web hat sich der Umgang mit Daten grundlegend geändert. Während bisher in erster Linie der Zugriff auf möglichst gut modellierte lokale Informationen ausschlaggebend war, rückt nun immer mehr der ad-hoc-Zugriff auf wenig strukturierte und verteilt vorliegende Informationen in den Vordergrund. Neuere Arbeiten zum Thema Datenintegration befassen sich wieder ausschließlich mit den statischen, strukturellen Eigenschaften ohne dynamische Aspekte und Kapselung. Ein Großteil der verfügbaren Informationen liegt jetzt nicht mehr in starren Schemas vor, sondern muß aus unregelmäßigen und tief verschachtelten oder auch völlig unstrukturierten Quellen gewonnen werden. Durch die neue Qualität dieser sogenannten *semistrukturierten Daten* entstehen neue Anforderungen in unterschiedlichen Bereichen ([ABS00]).

Als erstes braucht man eine geeignete Darstellung für unregelmäßige und teilweise tief verschachtelte Daten. Im Gegensatz zu relationalen Tabellen werden hier eher baum- oder netzartige Graphstrukturen verwendet, ähnlich zu strukturell objektorientierten Datenbanken. Für diese neuen Strukturen benötigt man weiterhin eine geeignete persistente Datenhaltung sowie eine Anfragesprache zum Zugriff auf die Daten ([Abi97]).

Ein erster Ansatz zur Darstellung semistrukturierter Daten ist das 1995 veröffentlichte OEM-Modell[48] der Stanford University ([PGW95]). Es beschreibt die Daten als gerichteter Graph mit Knoten und benannten Kanten. Der zugehörige Datenbankprototyp LORE wird in [MAG*97] beschrieben, eine Einführung in die Anfragesprache LOREL findet man in [AQH*97].

Von größerer Bedeutung ist zwischenzeitlich aber die aus SGML und HTML weiterentwickelte Auszeichnungssprache XML ([W21], [GoP02]), die als weltweit akzeptierte Standardfamilie vorliegt und immer mehr auch von der Industrie verwendet wird; beispielsweise in der Literatur oder im Portal- und Finanzsektor (*Dublin Core* [W22] bzw. *XML/EDI* [W23]). Für die Ablage von XML-Daten sind sowohl *native* Datenbanken verfügbar, die die XML-Strukturen direkt persistent ablegen, als auch Abbildungsverfahren auf relationale oder objektrelationale Datenbanken; einen guten Überblick gibt hier die Webseite von Ronald Bourret ([W24]). Für Anfragen auf

48. Object Exchange Model

XML sind im Laufe der Zeit unterschiedliche Ansätze verfolgt worden, wie etwa XSLT, XML-QL, XQL, Kweelt, XQuery, usw., eine Übersicht findet man etwa in [BoC00]; der neueste XQuery-Standard ist in [W25] beschrieben.

Ausgehend von den neuen Datenstrukturen ergeben sich auch neue Aspekte für Sichten und Integrationen. Das in [AGM*97] präsentierte Sichtsystem für LORE legt dabei besonderen Wert auf strukturelle Umformungen ohne ein verfügbares Schema, wobei durch sogenannte *Record Identifier* eine gewisse Objektidentität beibehalten werden kann ([ZhG98]). In [CaA99] wird ein Sichtsystem auf Basis von Ontologien vorgestellt und [LiD00] betrachtet semistrukturierte Sichten im Bereich von Data Warehouses. Im XML-Standard sind bisher noch keine Sichten vorgesehen; zwei Kommentare für mögliche Ansätze findet man unter [Abi99] und [BMR99]. Weitere Arbeiten befassen sich mit spezifischen Aspekten möglicher XML-Sichten ([StF02], [KLS03], [KSM03], [RCDF03]).

Die Integrationsverfahren für semistrukturierte Daten werden in erster Linie durch die Schemalosigkeit und Flüchtigkeit verfügbarer Informationen im Internet beeinflußt. Da eine aufwendige Integrationsphase selten lohnenswert ist, handelt es sich fast immer um ad-hoc-Verfahren mit graphischer Unterstützung. Um den Benutzer bei der Integration zu unterstützen, wird in vielen Fällen versucht, die Ursprungsdaten auf semantischer Ebene zu interpretieren, indem Ontologien oder Verfahren des Data Mining angewandt werden, um Originalquellen anzureichern ([SBH02]). Aus diesem Grund ist hier der Begriff der *Informationsintegration* geprägt worden ([Hea*98]). Ein häufig anzutreffendes Konzept bilden dabei die Mediatorsysteme: Die Architekturunabhängigkeit von XML ausnutzend werden Wrapper für unterschiedliche Datenquellen definiert, die von einem zentralen Mediator Anfragen in einer der XML-Anfragesprachen erwarten, sie intern auswerten, und die Ergebnisse in Form von XML-Daten zur weiteren Verarbeitung an den Mediator zurückliefern ([GMDS02], [BBC*02], [CCS00], [CKK03]). Eine wichtige Initiative in Richtung der Informationsintegration besteht darin, möglichst viel semantische Information bereits in den Datenquellen zu repräsentieren. Zu diesem *Semantic Web* haben bereits diverse Workshops und eine eigene Konferenz stattgefunden ([HoH02], [CKDE03]). Ein wichtiger Standard im XML-Umfeld ist dabei die *Web Ontology Language OWL* ([W26]), die sich als Basis für die generische Spezifikation von Integrationen einsetzen läßt ([LeF04]).

Literaturverzeichnis

[AAC*95] Jurgen Annevelink, Rafiul Ahad, Amelia Carlson, Daniel H. Fishman, Michael L. Heytens, William Kent: *Object SQL – A Language for the Design and Implementation of Object Databases;* In: Won Kim (Hrsg.): Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press und Addison-Wesley, 1995, ISBN 0–201–59098–0; S. 42–68

[AAD*93] Rafi Ahmed, Joseph Albert, Weimin Du, William Kent, Witold Litwin, Ming-Chien Shan: *An Overview of Pegasus;* In: Hans-Jörg Schek, Amit P. Sheth, Bogdan D. Czejdo (Hrsg.): Proc. 3rd Intl. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems RIDE–IMS'93 (19.–20. April 1993, Wien, Österreich), IEEE Computer Society, 1993, ISBN 0–8186–3710–2; S. 273–277

[AAG*01] Rahim Adatia, Faiz Arni, Kyle Gabhart, John Griffin, Matjaz B. Juric, Jeremiah Lott, Tim McAllister, Aaron Mulder, Nathan Nagarajan, Dan O'Connor, Ted Osborne, P. G. Sarang, Andre Tost, Dave Young, Craig A. Berry: *Professional EJB;* Wrox Press, Birmingham, 2001, ISBN 1–861005–08–3

[AbB91] Serge Abiteboul, Anthony J. Bonner: *Objects and Views;* In: James Clifford, Roger King (Hrsg.): Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data (29.–31. Mai 1991, Denver, USA), SIGMOD Record 20(2), Juni 1991; S. 238–247

[ABC*83] Malcolm P. Atkinson, Peter J. Bailey, Kenneth Chisholm, W. Paul Cockshott, Ronald Morrison: *An Approach to Persistent Programming;* In: *The Computer Journal*, 26(4), 1983; S.360–365

[ABDS94] Eric Amiel, Marie-Jo Bellosta, Eric Dujardin, Eric Simon: *Supporting Exceptions to Schema Consistency to Ease Schema Evolution in OODBMS;* In: Jorge B. Bocca, Matthias Jarke, Carlo Zaniolo (Hrsg.): Proc. 20th Intl. Conf. on Very Large Data Bases VLDB'94 (12.–15. September 1994, Santiago de Chile, Chile), Morgan Kaufmann, San Francisco, 1994, ISBN 1–55860–153–8; S. 108–119

[ABF02] Paulo Sérgio Almeida, Carlos Baquero, Victor Fonte: *Version Stamps: Decentralized Version Vectors;* Proc. 22nd Intl. Conf. on Distributed Computing Systems ICDCS'02 (2.–5. Juli 2002, Wien, Österreich)

[AbH87] Serge Abiteboul, Richard Hull: *IFO: A Formal Semantic Database Model;* ACM Transactions on Database Systems (TODS) 12(4), Dezember 1987; S. 525–565

[Abi90] Serge Abiteboul: *Virtuality in Object-Oriented Databases*; Preliminary report, INRIA, Frankreich, 1990

[Abi97] Serge Abiteboul: *Querying Semi-Structured Data;* In: Foto N. Afrati, Phokion Kolaitis (Hrsg.): Proc. 6th Intl. Conf. on Database Theory ICDT'97 (8.–10. Januar 1997, Delphi, Griechenland), LNCS 1186, Springer, Berlin, 1997, ISBN 3–540–62222–5; S. 1–18

[Abi99] Serge Abiteboul: *On Views and XML;* Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS'99 (31. Mai – 2. Juni 1999, Philadelphia, USA), ACM Press, 1999, ISBN 1–58113–062–7; S. 1–9

[AbK98] Serge Abiteboul, Paris C. Kanellakis: *Object Identity as a Query Language Primitive;* Journal of the ACM 45(5), September 1998; S. 798–842

[ABS00] Serge Abiteboul, Peter Buneman, Dan Suciu: *Data on the Web: From Relations to Semistructured Data and XML;* Morgan Kaufmann, San Francisco, 2000, ISBN 1–55860–622–X; 258 S.

[ABW*89] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, Stanley B. Zdonik: *The Object-Oriented Database System Manifesto;* In: Won Kim, Jean-Marie Nicolas, Shojiro Nishio (Hrsg.): Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases DOOD'89 (4.–6. Dezember 1989, Kyoto, Japan), North-Holland/Elsevier, 1989; sowie in [BDK92]

[AEA92] Divyakant Agrawal, Amr El Abbadi: *A Non-Restrictive Concurrency Control Protocol for Object-Oriented Databases;* In: Alain Pirotte, Claude Delobel, Georg Gottlob (Hrsg.): Advances in Database Technology – Proc. 3rd Intl. Conf. on Extending Database Technology EDBT'92 (23.–27. März 1992, Wien, Österreich), LNCS 580, Springer, Berlin, 1992; S. 469–482

[AGF03] Mohamed Al-Mourad, W. A. Gray, N. J. Fiddian: *MVMBS: A Multiple Views Supporting Multiple Behaviours System for Interoperable Object-Oriented Database Systems;* In: Anne E. James, Brian Lings, Muhammad Younas (Hrsg.): New Horizons in Information Management, Proc. 20th British Nat. Conf. on Databases BNCOD 20 (15.–17. Juli 2003, Coventry, UK), LNCS 2712, Springer, Berlin, 2003, ISBN 3–540–40536–4; S. 198–206

[AGM*97] Serge Abiteboul, Roy Goldman, Jason McHugh, Vasilis Vassalos, Yue Zhuge: *Views for Semistructured Data;* In: Dan Suciu (Hrsg.): Proc. Workshop on Management of Semistructured Data (in conjunction with SIGMOD/PODS, Mai 1997, Tucson, USA), 1997

[Ala99] Suad Alagic: *O_2 and the ODMG Standard: Do They Match?*; TAPOS – Theory and Practice of Object Systems 5(4), 1999; S. 239–247

[AlE99] Reda Alhajj, Ashraf Elnagar: *Incremental Materialization of Object-Oriented Views;* Data & Knowledge Engineering 29(2), Februar 1999, Elsevier Science B.V.; S. 121–145

[AlP96] Reda Alhajj, Faruk Polat: *View Maintenance in Object-Oriented Databases;* In: Roland Wagner, Helmut Thoma (Hrsg.): Proc. 7th Intl. Conf. on Database and Expert Systems Applications DEXA'96 (9.–13. September 1996, Zürich, Schweiz), LNCS 1134, Springer, Berlin, 1996, ISBN 3–540–61656–X; S. 153–162

[ANSI75] *ANSI/X3/SPARC Study Group on Data Management SystemS.* Interim Report; FDT (ACM SIGMOD RECORD) 7(2), 1975

[AQH*97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, Janet L. Wiener: *The Lorel Query Language for Semistructured Data;* Intl. Journal on Digital Libraries 1(1), 1997; S. 68–88

[BaK90] François Bancilhon, Won Kim: *Object-Oriented Database Systems: In Transition;* IEEE Data Engineering Bulletin 13(4), Dezember 1990; S. 24–28

[BaM95] Lars Bækgaard, Leo Mark: *Incremental Computation of Nested Relational Query Expressions;* ACM Transactions on Database Systems 20(2), Juni 1995; S. 111–148

[BaW95] Elena Baralis, Jennifer Widom: *Using Delta Relations to Optimize Condition Evaluation in Active Databases;* In: Timos Sellis (Hrsg.): Proc. 2. Intl. Workshop on Rules in Database Systems RIDS'95 (25.–27. September 1995, Athen, Griechenland), LNCS 985, Springer, Berlin, 1995, ISBN 3–540–60365–4; S. 292–308

[BBC*02] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Valeria De Antonellis, Alfio Ferrara, Francesco Guerra, Federica Mandreoli, Giorgio Carlo Ornetti, Maurizio Vincini: *Semantic Integration and Query Optimization of Heterogeneous Data Sources;* In: Jean-Michel Bruel, Zohra Bellahsene (Hrsg.): Proc. Workshop on Advances in Object-Oriented Information Systems OOIS'02 (2. September 2002, Montpellier, Frankreich), LNCS 2426, Springer, Berlin, 2002, ISBN 3–540–44088–7; S. 154–165

[BCD89] François Bancilhon, Sophie Cluet, Claude Delobel: *A Query Language for the O_2 Object-Oriented Database System*; In: Richard Hull, Ron Morrison, David Stemple (Hrsg.): Proc. 2nd Intl. Workshop on Database Programming Languages (4.–8. Juni 1989, Gleneden Beach, USA), Morgan Kaufman, San Mateo, 1990

[BDK92] François Bancilhon, Claude Delobel, Paris C. Kanellakis (Hrsg.): *Building an Object-Oriented Database System – The Story of O_2*; Morgan Kaufmann, San Mateo, 1992, ISBN 1–55860–169–4

[Bee89] Catriel Beeri: *Formal models for object-oriented databases;* In: Won Kim, Jean-Marie Nicolas, Shojiro Nishio (Hrsg.): Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases DOOD'89 (4.–6. Dezember 1989, Kyoto, Japan), North-Holland/Elsevier, 1989

[Bee90] Catriel Beeri: *A Formal Approach to Object-Oriented Databses;* Data and Knowledge Engineering 5(4), 1990

[BeG95] Elisa Bertino, Giovanna Guerrini: *Objects with Multiple Most Specific Classes;* In: Walter G. Olthoff (Hrsg.): Proc. 9th European Conf. on Object-Oriented Programming ECOOP'95 (7.–11. August 1995, Århus, Dänemark), LNCS 952, Springer, Berlin, 1995, ISBN 3–540–60160–0; S. 102–126

[Bel96] Zohra Bellahsene: *View Mechanism for Schema Evolution;* In: Ronald Morrison, Jessie B. Kennedy (Hrsg.): Advances in Databases, Proc. 14th British Nat. Conf. on Databases BNCOD 14 (3.–5. Juli 1996, Edinburgh, UK), LNCS 1094, Springer, Berlin, 1996, ISBN 3–540–61442–7; S. 18–35

[Bel97] Zohra Bellahsene: *Extending a View Mechanism to Support Schema Evolution in Federated Database Systems;* In: Abdelkader Hameurlain, A Min Tjoa (Hrsg.): Proc. 8th Intl. Conf. on Database and Expert Systems Applications DEXA'97 (1.–5. September 1997, Toulouse, Frankreich), LNCS 1308, Springer, Berlin, 1997, ISBN 3–540–63478–9; S. 573–582

[Bel00] Zohra Bellahsene: *Updates and Object-Generating Views in ODBS;* Data & Knowledge Engineering 34(2), August 2000; S. 125–163

[Ber91] Elisa Bertino: *Views in Object-Oriented Databases;* Technical Report, Universität Genf, 1991

[Ber92] Elisa Bertino: *A View Mechanism for Object-Oriented Databases*; In: Alain Pirotte, Claude Delobel, Georg Gottlob (Hrsg.): Advances in Database Technology – Proc. 3rd Intl. Conf. on Extending Database Technology EDBT'92 (23.–27. März 1992, Wien, Österreich), LNCS 580, Springer, Berlin, 1992, ISBN 3–540–55270–7; S. 136–151

[BFHK94] Ralph Busse, Peter Fankhauser, Gerald Huck, Wolfgang Klas: *IRO–DB: An Object-Oriented Approach Towards Federated and Interoperable DBMS;* Proc. 1st Intl. Workshop on Advances in Databases and Information Systems ADBIS'94 (23.–26. Mai 1994, Moskau, Rußland), Russian Academy of Sciences, Association for Computing Machinery, Moskau, 1994

[BFN94] Ralph Busse, Peter Fankhauser, Erich J. Neuhold: *Federated Schemata in ODMG;* In: Johann Eder, Leonid A. Kalinichenko (Hrsg.): East/West Database Workshop: Proc. 2nd Intl. East/West Database Workshop (25.–28. September 1994, Klagenfurt, Österreich), Workshops in Computing, Springer, London, 1995, ISBN 3-540-19946-2; S. 356–379

[BGS95] Yuri Breitbart, Hector Garcia-Molina, Abraham Silberschatz: *Transaction Management in Multidatabase Systems;* In: Won Kim (Hrsg.): Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press und Addison-Wesley, 1995, ISBN 0–201–59098–0; S. 573–591

[BKK88] Jay Banerjee, Won Kim, Kyung-Chang Kim: *Queries in Object-Oriented Databases;* Proc. 4th Intl. Conf. on Data Engineering ICDE'88 (1.–5. Februar 1988, Los Angeles, USA), IEEE Computer Society, 1988, ISBN 0–8186–0827–7; S. 31–38

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, Henry F. Korth: *Semantics and Implementation of Schema Evolution in Object-Oriented Databases;* In: Umeshwar Dayal, Irving L. Traiger (Hrsg.): Proc. 1987 ACM SIGMOD Intl. Conf. on Management of Data (27.–29. Mai 1987, San Francisco, USA), SIGMOD Record 16(3), Dezember 1987; S. 311–322

[BKSW91] Thierry Barsalou, Arthur M. Keller, Niki Siambela, Gio Wiederhold: *Updating Relational Databases through Object-Based Views;* In: James Clifford, Roger King (Hrsg.): Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data (29.–31. Mai 1991, Denver, USA), SIGMOD Record 20(2), Juni 1991; S. 248–257

[BLN86] Carlo Batini, Maurizio Lenzerini, Shamkant B. Navathe: *A Comparative Analysis of Methodologies for Database Schema Integration;* ACM Computing Surveys 18(4), Dezember 1986; S. 323–364

[BLT86] José A. Blakeley, Per-Åke Larson, Frank W. Tompa: *Efficiently Updating Materialized Views;* In: Carlo Zaniolo (Hrsg.): Proc. Intl. Conf. on Management of Data SIGMOD'86 (28.–30. Mai 1986, Washington D.C., USA), Association for Computing Machinery, New York, 1986, ISBN 0–89791–191–1; S. 61–71

[BMR99] David Beech, Ashok Malhotra, Michael Rys: *A Formal Data Model and Algebra for XML;* W3C XML Query working group note, September 1999

[BNPS89] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, Licia Sbattella: *Integration of Heterogeneous Database Applications Through an Object-Oriented Interface*; Information Systems 14(5), Pergamon Press, Oxford, 1989; S. 407–420

[BNPS94] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, Licia Sbattella: *Applications of Object-Oriented Technology to the Integration of Heterogeneous Database Systems;* Distributed and Parallel Databases 2(4), Oktober 1994, Kluwer Academic Publishers, ACM SIGMOD; S. 343–370

[BoC00] Angela Bonifati, Stefano Ceri: *Comparative Analysis of Five XML Query Languages;* SIGMOD Record 29(1), März 2000; S. 68–79

[BuF98] Ralph Busse, Peter Fankhauser: *Declarative and Procedural Object-Oriented Views GMD Report 38*; GMD – Forschungszentrum Informationstechnik GmbH, Sankt Augustin, Oktober 1998, ISSN 1435–2702; 19 S.

[BuF99] Ralph Busse, Peter Fankhauser: *Declarative and Procedural Object-Oriented Views*; In: Masaru Kitsuregawa, Leszek Maciaszek, Mike Papazoglou, Calton Pu (Hrsg.): Proc. 15th Intl. Conf. on Data Engineering ICDE'99 (23.–26. März 1999, Sydney, Australien), IEEE Computer Society, Los Alamitos, 1999; S. 260

[Bus91] Ralph Busse: E*inordnung von Anfrageergebnissen der Objektalgebra in einen Klassenverband und ihre Implementierung;* Diplomarbeit, Institut für Informatik, TU Clausthal, 1991

[Bus96] Ralph Busse: *IRO–DB C++ Binding Specification;* Technical Note IRO/TN/GMD/BINDING–V1.3/RB960315, IRO–DB Project P8629, ESPRIT–III Programme, März 1996

[ByM93] Kwang June Byeon, Dennis McLeod: *Towards the Unification of Views and Versions for Object Databases*; In: Shojiro Nishio, Akinori Yonezawa (Hrsg.): Object Technologies for Advanced Software, Proc. 1st JSSST Intl. Symp. ISOTAS'93 (4.–6. November 1993, Kanazawa, Japan), LNCS 742, Springer, Berlin, 1993, ISBN 3–540–57342–9; S. 220–236

[CaA99] Silvana Castano, Valeria De Antonellis: *Building Views over Semistructured Data Sources;* In: Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, Elisabeth Métais (Hrsg.): Proc. 18th Intl. Conf. on Conceptual Modeling ER'99 (15.–18. November 1999, Paris, Frankreich), LNCS 1728, Springer, Berlin, 1999, ISBN 3–540–66686–9; S. 146–160

[CaB97] Rick Cattell, Douglas Barry (Hrsg.): *The Object Database Standard: ODMG 2.0;* Morgan Kaufman, San Francisco, 1997, ISBN 1-55860-463-4

[CaB00] Rick Cattell, Douglas Barry (Hrsg.): *The Object Data Standard: ODMG 3.0;* Morgan Kaufman, San Francisco, 2000, ISBN 1-55860-647-5

[CaF92] Michèle Cart, Jean Ferrié: *Integrating Concurrency Control;* In: [BDK92], S. 463–485

[Cat93] Rick Cattell (Hrsg.): *The Object Database Standard: ODMG–93;* Morgan Kaufman, San Francisco, 1993

[Cat94] Rick Cattell (Hrsg.): *The Object Database Standard: ODMG–93;* Release 1.1, Morgan Kaufman, San Francisco, 1994, ISBN 1-55860-302-6

[Cat96] Rick Cattell (Hrsg.): *The Object Database Standard: ODMG–93;* Release 1.2, Morgan Kaufman, San Francisco, 1996, ISBN 1-55860-396-4

[CBR94] Armin B. Cremers, Oleg T. Balovnev, Wolfgang Reddig: *Views in Object-Oriented Databases;* Proc. 1st Intl. Workshop on Advances in Databases and Information Systems ADBIS'94 (23.-26. Mai 1994, Moskau, Rußland), Russian Academy of Sciences, Association for Computing Machinery, Moskau, 1994; S. 74–82

[CCS00] Vassilis Christophides, Sophie Cluet, Jérôme Siméon: *On Wrapping Query Languages and Efficient XML Integration;* In: Weidong Chen, Jeffrey F. Naughton, Philip A. Bernstein (Hrsg.): Proc. 2000 ACM SIGMOD Intl. Conf. on Management of Data (16.–18. Mai 2000, Dallas, USA), SIGMOD Record 29(2), ACM, 2000, ISBN 1–58113–218–2; S. 141–152

[ChB99] Yangjun Chen, Wolfgang Benn: *Integrating Heterogeneous OO Schemas;* In: Masaru Kitsuregawa, Leszek Maciaszek, Mike Papazoglou, Calton Pu (Hrsg.): Proc. 15th Intl. Conf. on Data Engineering ICDE'99 (23.–26. März 1999, Sydney, Australien), IEEE Computer Society, Los Alamitos, 1999; S. 101 (Langfassung des Beitrags von den Autoren verfügbar.)

[CHKS92] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, Pierangela Samarati: *Achieving Incremental Consistency among Autonomous Replicated Databases;* In: David K. Hsiao, Erich J. Neuhold, Ron Sacks-Davis (Hrsg.): Proc. Intl. Workshop on the Semantics of Interoperable Database Systems DS–5 (16.–20. November 1992, Lorne, Australien), IFIP International Federation for Information Processing, Lorne, 1992; S. 223–237

[ChL92] Jan Chomicki, Witold Litwin: *Declarative Definition of Object-Oriented Multidatabase Mappings;* In: M. Tamer Özsu, Umeshwar Dayal, Patrick Valduriez (Hrsg.): Distributed Object Management, Papers from the Intl. Workshop on Distributed Object Management IWDOM'92 (19.–21. August 1992, Edmonton, Kanada), Morgan Kaufmann, San Francisco, 1994, ISBN 1–55860–256–9; S. 375–392

[CHM92] Vinay K. Chaudhri, Vassos Hadzilacos, John Mylopoulos: *Concurrency Control for Knowledge Bases;* In: Bernhard Nebel, Charles Rich, William R. Swartout (Hrsg.): Proc. 3rd Intl. Conf. on Principles of Knowledge Representation and Reasoning KR'92 (25.–29. Oktober 1992, Cambridge, USA). Morgan Kaufmann, San Mateo, 1992, ISBN 1–55860–262–3; S. 762–773

[ChS93] Qiming Chen, Ming-Chien Shan: *Abstract View Objects for Multiple OODB Integration;* In: Shojiro Nishio, Akinori Yonezawa (Hrsg.): Proc. 1st JSSST Intl. Symp. Object Technologies for Advanced Software ISOTAS'93 (4.–6. November 1993, Kanazawa, Japan), LNCS 742, Springer, Berlin, 1993, ISBN 3–540–57342–9; S. 237–250

[CKDE03] Isabel F. Cruz, Vipul Kashyap, Stefan Decker, Rainer Eckstein (Hrsg.): *Proc. 1st Intl. Workshop on Semantic Web and Databases SWDB'03 (7.–8. September 2003, Berlin, Deutschland)*, co-located with VLDB 2003, Humbold-Universität Berlin, 2003

[CKK*03] Vassilis Christophides, Gregory Karvounarakis, I. Koffina, G. Kokkinidis, Aimilia Magkanaraki, Dimitris Plexousakis, G. Serfiotis, Val Tannen: *The ICS–FORTH SWIM: A Powerful Semantic Web Integration Middleware;* In: Isabel F. Cruz, Vipul Kashyap, Stefan Decker, Rainer Eckstein (Hrsg.): Proc. 1st Intl. Workshop on Semantic Web and Databases SWDB'03 (7.–8. September 2003, Berlin, Deutschland), co-located with VLDB 2003, Humbold-Universität Berlin, 2003; S. 381–393

[CKZ94] Inessa Chaban, Leonid Kalinichenko, Vladimir Zadorozhny: *Could the OODB Standards be Better if More Grounded?;* In: Johann Eder, Leonid A. Kalinichenko (Hrsg.): Proc. 2nd Intl. East/West Database Workshop (25.–28. September 1994, Klagenfurt, Österreich), Workshops in Computing, Springer, London, 1995, ISBN 3-540-19946-2; S. 232–248

[Cod70] Edgar F. Codd: *A relational model for large shared data banks;* Communications of the ACM 13(6), Juni 1970; S. 377–587

[Cod79] Edgar F. Codd: *Extending the Database Relational Model to Capture More Meaning;* ACM Transactions on Database Systems 4(4), Dezember 1979

[CRS*92] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, Walter Zimmer: *OBST – An Overview;* STONE–Report FZI.039.1, Forschungszentrum Informatik (FZI), Haid-und-Neu-Straße 10–14, D–76131 Karlsruhe, Juni 1992

[DaD98] Hugh Darwen, Chris J. Date: *SQL – der Standard: SQL/92 mit den Erweiterungen CLI und PSM;* Addison-Wesley, Bonn, 1998, ISBN 3–8273–1345–7; 582 S.

[Dat86] C. J. Date: *An Introduction to Database Systems;* 4. Ausg., Addison Wesley, Reading, Mass., 1986, ISBN 0-20119215-2

[Day89] Umeshwar Dayal: *Queries and Views in an Object-Oriented Data Model;* In: Richard Hull, Ronald Morrison, David W. Stemple (Hrsg.): Proc. 2nd Intl. Workshop on Database Programming Languages DBPL'89 (4.–8. Juni 1989, Gleneden Beach, USA), Morgan Kaufmann, San Francisco, 1990, ISBN 1–55860–072–8; S. 80–102

[DDK*96] Asuman Dogaç, Cevdet Dengi, Ebru Kilic, Gökhan Ozhan, Fatma Ozcan, Sena Nural, Cem Evrendilek, Ugur Halici, Budak Arpinar, Pinar Koksal, Sema Mancuhan: *A Multidatabase System Implementation on CORBA*; Proc. 6th Intl. Workshop on Research Issues in Data Engineering RIDE-NDS'96 (26.–27. Februar 1996, New Orleans, USA), IEEE Computer Society, Los Alamitos, 1996; S. 2–11

[DDÖ98] Asuman Dogaç, Cevdet Dengi, M. Tamer Özsu: *Distributed Object Computing Platforms;* Communications of the ACM 41(9), September 1998; S. 95–103

[DeZ91] Christine Delcourt, Roberto Zicari: *The Design of an Integrity Consistency Checker (ICC) for an Object-Oriented Database System;* In: Pierre America (Hrsg.): Proc. Europ. Conf. on Object-Oriented Programming ECOOP'91 (15.–19. Juli 1991, Genf, Schweiz), LNCS 512, Springer, Berlin, 1991, ISBN 3–540–54262–0; S. 97–117

[DFG96] Rehab M. Duwairi, N. J. Fiddian, W. A. Gray: *Schema Integration Meta-Knowledge Classification and Reuse;* In: Ron Morrsion, Jessie Kennedy (Hrsg.): Proc. 14th British National Conf. on Databases BNCOD 14 (3.–5. Juli 1996, Edinburgh, UK), LNCS 1094, Springer, Berlin, 1996, ISBN 3–540–61442–7; S. 1–17

[Dob95] Michael Dobrovnik: *Externe Schemata in objektorientierten Datenbanksystemen;* Dissertation am Institut für Informatik, Universität Klagenfurt, August 1995; veröffentlicht als: DISDBIS 25, Infix Verlag, St. Augustin 1997, ISBN 3–89601–425–0

[DoE93a] Michael Dobrovnik, Johann Eder: *View Concepts for Object-Oriented Databases;* In: G. E. Lasker (Hrsg.): Advances in Cognitive Engineering and Knowledge-Based Systems, Proc. 4th Intl. Symp. Systems Research, Informatics and Cybernetics, (August 1993, Baden-Baden); S. 163–167

[DoE93b] Michael Dobrovnik, Johann Eder: *A Concept of Type Derivation for Object-Oriented Database Systems;* In: L. Gün, R. Onurval, E. Gelenbe (Hrsg.): Proc. 8th Intl. Symp. Computer and Information Sciences ISCIS VIII (3.–5. November 1993, Antalya, Türkei), 1993; S. 113-120

[DoE94] Michael Dobrovnik, Johann Eder: *Adding View Support to ODMG–93;* Proc. 1st Intl. Workshop on Advances in Databases and Information Systems ADBIS'94 (23.-26. Mai 1994, Moskau, Rußland), Russian Academy of Sciences, Association for Computing Machinery, Moskau, 1994; S. 62–73

[DoE96] Michael Dobrovnik, Johann Eder: *Logical Data Independence and Modularity Through Views in OODBMS;* Proc. Intl. Conf. Engineering Systems Design and Analysis – Object-Oriented Database Management Systems ESDA'96 (1996, Montpellier, Frankreich), 1996; Band 2, S. 13–20

[DoE97] Michael Dobrovnik, Johann Eder: *Derived Types and Method Steadiness in Updateable Views;* Universität Klagenfurt, Mai 1997; Quelle unbekannt

[DoE98] Michael Dobrovnik, Johann Eder: *Partial Replication of Object-Oriented Databases;* In: Witold Litwin, Tadeusz Morzy, Gottfried Vossen (Hrsg.): Advances in Databases and Information Systems, Proc. 2nd East European Symposium ADBIS'98 (7.–10. September 1998, Poznan, Polen), LNCS 1475, Springer, Berlin, 1998, ISBN 3–540–64924–7; S. 260–271

[Dor97] Rowan Dordick: *The Secrets of Agents;* IBM Research Magazine, 35(1), 1997

[Duw97] Rehab M. Duwairi: *Views for Interoperability in a Heterogeneous Object-Oriented Multidatabase System;* PhD Thesis, Department of Computer Science, University of Wales, College of Cardiff, April 1997

[EdF94] Johann Eder, Heinz Frank: *Schema Integration for Object Oriented Database Systems;* In: M. Tanik, W. Rossak, D. Cooke (Hrsg.): Software Systems in Engineering, ASME, PD–Vol. 59, 1994; S. 275–284

[EGLT76] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, Irving L. Traiger: *The Notions of Consistency and Predicate Locks in a Database System;* Communications of the ACM 19(11), November 1976, Association for Computing Machinery, New York; S. 624–633

[EiM99] Andrew Eisenberg, Jim Melton: *SQL: 1999, formerly known as SQL 3;* SIGMOD Record 28(1), März 1999; S. 131–138

[EKPT96] Wolfgang Eßmayr, Fritz Kastner, Günther Pernul, A Min Tjoa: *The Security Architecture of IRO–DB*; Proc. 12th IFIP Intl. Conf. on Information Security, IFIP/SEC 96, 1996

[EKP*96] Wolfgang Eßmayr, Fritz Kastner, Günther Pernul, Stefan Preishuber, A Min Tjoa: *Authorization and Access Control in IRO–DB*; In: Stanley Y. W. Su (Hrsg.): Proc. 12th Intl. Conf. on Data Engineering ICDE'96 (26. Februar – 1. März 1996, New Orleans, USA), IEEE Computer Society Press, Washington, 1996, ISBN 0–8186–7240–4

[ElS92] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual;* Addison-Wesley, Reading, 1992, ISBN 0–201–51459–1; 453 S.

[Emp97] Arnd Empting: *Entwurf und Implementierung eines Objekt-Caches auf CORBA*; Diplomarbeit, Fachhochschule Darmstadt, März 1997

[Fan97] Peter Fankhauser: *A Methodology for Knowledge-Based Schema Integration*; PHD Thesis, Technische Universität Wien, Österreich, Dezember 1997

[FaV95] Christian Fahrner, Gottfried Vossen: *Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG–93*; In: Tok Wang Ling, Alberto O. Mendelzon, Laurent Vieille (Hrsg.): Proc. 4th Intl. Conf. on Deductive and Object-Oriented Databases DOOD'95 (4.–7. Dezember 1995, Singapur), LNCS 1013, Springer, Berlin, 1995, ISBN 3–540–60608–4; S. 429–446

[FBA*90] Daniel H. Fishman, David Beech, Jurgen Annevelink, E. C. Chow, Tim Connors, J. W. Davis, Waqar Hasan, C. G. Hoch, William Kent, S. Leichner, Peter Lyngbæk, Brom Mahbod, Marie-Anne Neimat, Tore Risch, Ming-Chien Shan, W. Kevin Wilkinson: *Overview of the Iris DBMS;* In: Alfonso F. Cardenas, Dennis McLeod (Hrsg.): Research Foundations in Object-Oriented and Semantic Database Systems; Prentice-Hall, 1990, ISBN 0–13–806340–0; S. 174–199

[FFK96] Peter Fankhauser, Béatrice Finance, Wolfgang Klas: *IRO–DB: Making Relational and Object-Oriented Databases Interoperable*; In: Peter M. G. Apers, Mokrane Bouzeghoub, Georges Gardarin (Hrsg.): Advances in Database Technology, Proc. 5th Intl. Conf. on Extending Database Technology EDBT'96 (25.–29. März 1996, Avignon, Frankreich), LNCS 1057, Springer, Berlin, 1996, ISBN 3-540-61057-X; S. 485–489

[FGMS93] Doug Fang, Shahram Ghandeharizadeh, Dennis McLeod, Antonio Si: *The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database Systems;* In: Proc. 9th Intl. Conf. on Data Engineering ICDE'93 (19.–23. April 1993, Wien, Österreich), IEEE Computer Society, 1993, ISBN 0–8186–3570–3; S. 467–475

[Fla97] David Flanagan: *Java in a Nutshell, Second Edition*; O'Reilly & Associates, Inc., Sebastopol, CA, 1997, ISBN 1–56592–262–X

[FNLS88] Peter Fankhauser, Erich J. Neuhold, Witold Litwin, Michael Schrefl: *Global View Definition and Multi-database Languages – Two Approaches to Database Integration;* In: Rolf Speth (Hrsg.): Proc. European Teleinformatics Conf. on Research into Networks and Distributed Applications EUTECO'88 (20.–22. April 1988, Wien, Österreich), North-Holland, Amsterdam, 1988; S. 1069–1082

[Fra96] Michael J. Franklin: *Client Data Caching: A Foundation for High Performance Object Database Systems;* Kluwer intl. series in engineering and computer science, Nr. 354, Kluwer Academic Publishers, Boston, 1996, ISBN 0–7923–9701–0

[FRH97] Fernando de Ferreira Rezende, Theo Härder: *Exploiting Abstraction Relationships' Semantics for Transaction Synchronization in KBMSs;* Data & Knowledge Engineering 22(3), Mai 1997, North Holland, Amsterdam; S. 233–259

[FSF95] Béatrice Finance, Véronique Smahi, Jérôme Fessy: *Query Processing in IRO–DB*; Proc. 4th Intl. Conf. on Deductive and Object-Oriented Databases DOOD'95 (4.–7. Dezember 1995, Singapur), LNCS 1013, Springer, Berlin, 1995, ISBN 3–540–60608–4

[GFF97] Georges Gardarin, Béatrice Finance, Peter Fankhauser: *Federating Object-Oriented and Relational Databases: The IRO–DB Experience*; Proc. 2nd Intl. Conf. on Cooperative Information Systems CoopIS'97 (Kiawah Island, South Carolina), IEEE Computer Society, Los Alamitos, 1997

[GGF*96] Georges Gardarin, Sofiane Gannouni, Béatrice Finance, Peter Fankhauser, Wolfgang Klas, Dominique Pastre, Régis Legoff, Antonis Ramfos: *IRO-DB: A Distributed System Federating Object and Relational Databases;* In: Omran A. Bukhres, Ahmed K. Elmagarmid (Hrsg.): Object Oriented Multidatabase Systems: A solution for Advanced Applications, Prentice Hall; Englewood Cliffs, N.J., 1996, ISBN 0-13-103813-3; Kapitel 20

[GGT95] Georges Gardarin, Jean-Robert Gruser, Zhao-Hui Tang: *A Cost Model for Clustered Object-Oriented Databases*; In: Umeshwar Dayal, Peter M. D. Gray, Shojiro Nishio (Hrsg.): Proc. 21st Intl. Conf. on Very Large Data Bases VLDB'95 (11.–15. September 1995, Zürich, Schweiz), Morgan Kaufmann, 1995, ISBN 1–55860–379–4; S. 323–334

[GLPT76] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: *Granularity of Locks and Degrees of Consistency in a Shared Data Base;* In: G. M. Nijssen (Hrsg.): Proc. IFIP Working Conf. on Modelling in Data Base Management Systems (5.–8. Januar 1976, Freudenstadt), International Federation for Information Processing, North Holland, Amsterdam, Neuauflage 1978; S. 365–394

[GLRS93] John Grant, Witold Litwin, Nick Roussopoulos, Timos K. Sellis: *Query Languages for Relational Multidatabases;* VLDB Journal 2(2), 1993; S. 153–171

[GMDS02] Georges Gardarin, Antoine Mensch, Tuyet-Tram Dang-Ngoc, Lars Smit: *Integrating Heterogeneous Data Sources with XML and XQuery;* 13th Intl. Workshop on Database and Expert Systems Applications DEXA'2002 (2.–6. September 2002, Aix-en-Provence, Frankreich), IEEE Computer Society, 2002, ISBN 0–7695–1668–8; S. 839–846

[GOP96] GOPAS Software: *Ontos–ODB Library*, Software and Documentation, 1996

[GoP02] Charles F. Goldfarb, Paul Prescod: *The XML Handbook;* 4. Auflage, The Charles F. Goldfarb definitive XML series, Prentice Hall, Upper Saddle River, 2002, ISBN 0–13–065198–2; 1147 S.

[GoR85] Adele Goldberg, David Robson: *Smalltalk–80: The Language and its Implementation;* Addison-Wesley, Reading, Mass., 1985

[GRS91] Dimitrios Georgakopoulos, Marek Rusinkiewicz, Amit P. Sheth: *On Serializability of Multidatabase Transactions Through Forced Local Conflicts*; Proc. 7th Intl. Conf. on Data Engineering ICDE'91 (April 1991, Kobe, Japan), IEEE Computer Society Press, CA, 1991; S. 314–323

[GST96] Georges Gardarin, Fei Sha, Zhao-Hui Tang: *Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System*; In: T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, Nandlal L. Sarda (Hrsg.): Proc. 22nd Intl. Conf. on Very Large Data Bases VLDB'96 (3.–6. September 1996, Mumbai (Bombay), Indien), Morgan Kaufmann, San Francisco, 1996, ISBN 1–55860–382–4; S. 378–389

[HaH91] Thanasis Hadzilacos, Vassos Hadzilacos: *Transaction Synchronization in Object Bases;* Journal of Computer and System Siences 43(1), August 1991, Academic Press, Orlando, Florida; S. 2–24

[HaM81] Michael Hammer, Dennis McLeod: *Database Description with SDM: A Semantic Database Model;* ACM Transactions on Database Systems (TODS) 6(3), September 1981; S. 351–386

[HaR90] Stephen Hayne, Sudha Ram: *Multi-User View Integration System (MUVIS): An Expert System for View Integration;* Proc. 6th Intl. Conf. on Data Engineering ICDE'90 (5.–9. Februar 1990, Los Angeles, USA), IEEE Computer Society, 1990, ISBN 0–8186–2025–0; S. 402–409

[Hea*98] Marti A. Hearst et al.: *Trends & Controversies: Information Integration;* IEEE Intelligent Systems 13(5), September/Oktober 1998; S. 12–24

[HeB88] Sandra Heiler, Barbara T. Blaustein: *Generating and Manipulating Identifiers for Heterogeneous, Distributed Objects;* In: John Rosenberg, David Koch (Hrsg.): Proc. 3. Intl. Workshop on Persistent Object Systems POS'89 (10.–13. Januar 1989, Newcastle, Australien), Workshops in Computing, Springer, Berlin, 1989, ISBN 3–540–19626–9; S. 235–247

[HeS91] Andreas Heuer, Peter Sander: *Classifying Object-Oriented Query Results in a Class/Type Lattice;* In: Bernhard Thalheim, János Demetrovics, Hans-Detlef Gerhardt (Hrsg.): 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems MFDBS'91 (6.–9. Mai 1991, Rostock), LNCS 495, Springer, Berlin, 1991, ISBN 3–540–54009–1; S. 14–28

[HeS93] Andreas Heuer, Peter Sander: *The LIVING IN A LATTICE Rule Language;* Data & Knowledge Engineering 9(3), 1993; S. 249–286

[Heu87] Andreas Heuer: *Das IRIS-Projekt – Ein Systemüberblick;* Informatik-Bericht 87/1, Institut für Informatik, TU Clausthal, 1987

[Heu89] Andreas Heuer: *A Data Model for Complex Objects Based on a Semantic Database Model and Nested Relations;* In: Serge Abiteboul, Patrick C. Fischer, Hans-Jörg Schek (Hrsg.): Nested Relations and Complex Objects in Databases, LNCS 361, Springer, Berlin, 1989; S. 297–312

[Heu92] Andreas Heuer: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme;* Addison-Wesley, Bonn, 1992, ISBN 3–89319–315–4; 628 S.

[HeW88] Maurice P. Herlihy, William E. Weihl: *Hybrid Concurrency Control for Abstract Data Types;* Proc. 7th ACM Symposium on Principles of Database Systems PODS'88 (21.–23. März 1988, Austin, USA), ACM Press, New York, 1988, ISBN 0–89791–263–2; S. 201–210

[HeZ88a] Sandra Heiler, Stanley B. Zdonik: *FUGUE: A Model for Engineering Information Systems and Other Baroque Applications;* In: Catriel Beeri, Joachim W. Schmidt, Umeshwar Dayal (Hrsg.): Proc. 3. Intl. Conf. on Data and Knowledge Bases JCDKB'88: Improving Usability and Responsiveness (28.-30. Juni 1988, Jerusalem, Israel), Morgan Kaufmann, 1988, ISBN 0–93413–95–8; S. 195–210

[HeZ88b] Sandra Heiler, Stanley B. Zdonik: *Views, Data Abstraction, and Inheritance in the FUGUE Data Model;* In: Klaus R. Dittrich (Hrsg.): Advances in Object-Oriented Database Systems, Proc. 2. Intl. Workshop on Object-Oriented Database Systems OODBS'88 (27.–30. September 1988, Bad Münster), LNCS 334, Springer, Berlin, 1988, ISBN 3–540–50345–5; S. 225–241

[HeZ90] Sandra Heiler, Stanley Zdonik: *Object Views: Extending the Vision;* Proc. 6th Intl. Conf. on Data Engineering ICDE'90 (5.-9. Februar 1990, Los Angeles, USA), IEEE Computer Society Press, Los Alamitos, CA, 1990, ISBN 0–8186–2025–0; S. 86–93

[HFW90] Andreas Heuer, Jürgen Fuchs, Uwe Wiebking: *OSCAR: An Object-Oriented Database System with a Nested Relational Kernel;* In: Hannu Kangassalo (Hrsg.): Proc. 9th Intl. Conf. on Entity–Relationship Approach – The Core of Conceptual Modeling ER'90 (8.–10. Oktober 1990, Lausanne, Schweiz), North–Holland, Amsterdam, 1991, ISBN 0–444–88941–8; S. 95–110

[HöH91] Carsten Hörner, Andreas Heuer: *EXTREM – The Structural Part of an Object-Oriented Database Model;* Informatik-Bericht 91/5, Institut für Informatik, TU Clausthal, Clausthal–Zellerfeld, Oktober 1991

[HoH02] Ian Horrocks, James A. Hendler (Hrsg.): *The Semantic Web – Proc. 1st Intl. Semantic Web Conf. ISWC'02 (9.–12. Juni 2002, Sardinien, Italien),* LNCS 2342, Springer, Berlin, 2002, ISBN 3–540–43760–6; 476 S.

[Hul97] Richard Hull: *Managing Semantic Heterogeneity in Databases: A Theoretical Perspective;* Proc. 16th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems PODS'97 (12.–14. Mai 1997, Tucson, USA), ACM Press, 1997, ISBN 0–89791–910–6; S. 51–61

[IBM96] IBM: *Object Collection Service*; OMG Document Number 96.5.5, IBM submission paper, 1996

[JTTW88] John Joseph, Satish Thatte, Craig W. Thompson, David L. Wells: *Report on the Object-Oriented Database Workshop (in conjunction with OOPSLA'88) – Section 4: Panel on Schema Evolution and Version Management;* SIGMOD Record 18(3), September 1989; S. 90–95

[KAA*92] William Kent, Rafi Ahmed, Joseph Albert, Mohammad A. Ketabchi, Ming-Chien Shan: *Object Identification in Multidatabase Systems;* In: David K. Hsiao, Erich J. Neuhold, Ron Sacks-Davis (Hrsg.): Proc. IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems DS–5 (16.–20. November 1992, Lorne, Australien), IFIP Transactions A–25, North–Holland, Amsterdam, 1993, ISBN 0–444–89879–4; S. 313–330 (S. 302–319 in der Konferenzausgabe)

[KaW97] Elisabeth Kapsammer, Roland R. Wagner: *The IRO–DB Approach – Processing Queries in Federated Database Systems*; Proc. 1st Intl. Workshop on Query Processing and Multimedia Issues in Distributed Systems QPMIDS'97 (in conjunction with DEXA'97 Conference, September 1997, Toulouse, Frankreich), IEEE Society Press, 1997

[Kay94] M. Kay: *A Peg in the Ground: a Review of the Object Database Standard: ODMG–93*; In: First ClasS. The OMG Newsletter, 2(4), 1994; S. 13

[KDN90] Manfred Kaul, Klaus Drosten, Erich J. Neuhold: *ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views;* Proc. 6th Intl. Conf. on Data Engineering ICDE'90 (5.–9. Februar 1990, Los Angeles, USA), IEEE Computer Society, 1990, ISBN 0–8186–2025–0; S. 2–10

[Kel85] Arthur M. Keller: *Updating Relational Databases Through Views;* Ph.D. Dissertation, Stanford University, Calif., Department of Computer Science, Technical Report STAN-CS-85-1040, 1985

[Ken89] William Kent: *The Many Forms of a Single Fact;* Proc. 34th IEEE Computer Society Intl. Conf. COMPCON (27. Februar – 3. März 1989, San Francisco, USA), IEEE Computer Society, 1989; S. 438–443; auch: Technical Report HPL–SAL–88–8, Hewlett-Packard Laboratories, Oktober 1988

[KFA94] Wolfgang Klas, Gisela Fischer, Karl Aberer: *Integrating Relational and Object-Oriented Database Systems Using a Metaclass Concept;* Journal of Systems Integration 4(4), 1994, Kluwer Academic Publishers

[KFM*96] Wolfgang Klas, Peter Fankhauser, Peter Muth, Thomas Rakow, Erich J. Neuhold: *Database Integration using the Open Object-Oriented Database System VODAK*; In: Omran Bukhres, Ahmed K. Elmagarmid (Hrsg.): Object Oriented Multidatabase Systems: A Solution for Advanced ApplicationS. Prentice Hall, Englewood Cliffs, NJ, 1996; Kapitel 14

[KGK*95] William Kelley, Sunit Gala, Won Kim, Tom Reyes, Bruce Graham: *Schema Architecture of the UniSQL/M Multidatabase System;* In: Won Kim (Hrsg.): Modern Database Systems, ACM Press, New York, Addison Wesley, 1995, ISBN 0-201-59098-0; S. 621–648

[KiK95] Won Kim, William Kelley: *On View Support in Object-Oriented Database Systems;* In: Won Kim (Hrsg.): Modern Database Systems, ACM Press, New York, Addison Wesley, 1995, ISBN 0-201-59098-0; S. 108–129

[Kim89] Won Kim: *A Model of Queries for Object-Oriented Databases;* In: Peter M. G. Apers, Gio Wiederhold (Hrsg.): Proc. 15th Intl. Conf. on Very Large Data Bases VLDB'89 (22.–25. August 1989, Amsterdam, Niederlande), Morgan Kaufmann, 1989, ISBN 1–55860–101–5; S. 423–432

[Kim94a] Won Kim: *Observations on ODMG–93*; In: First ClasS. The OMG Newsletter, 2(4), 1994; S. 14–15

[Kim94b] Won Kim: *Observations on ODMG–93 Proposal*; SIGMOD Record 23(1), 1994; S. 4–9

[KiT91] Yutaka Kimura, Kunitoshi Tsuruoka: *A View Class Mechanism for Object-Oriented Database Systems;* In: Akifumi Makinouchi (Hrsg.): Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications DASFAA'91 (2.–4. April 1991, Tokio, Japan), Advanced Database Research and Development Series 2, World Scientific 1992, ISBN 981–02–1055–8; S. 269–273

[Kla92] Theo Klaustal: *Das OSCAR-Projekt;* Informatik-Bericht 92/2, Institut für Informatik, TU Clausthal, Clausthal-Zellerfeld, August 1992

[Kla93] Wolfgang Klas et al.: *VODAK Design Specification Document;* GMD–IPSI internal draft, September 1993

[KLS03] Hanna Kozankiewicz, Jacek Leszczylowski, Kazimierz Subieta: *Updatable XML Views;* In: Leonid A. Kalinichenko, Rainer Manthey, Bernhard Thalheim, Uwe Wloka (Hrsg.): Proc. 7th East European Conf. on Advances in Databases and Information Systems ADBIS 2003 (3.–6. September 2003, Dresden, Deutschland), LNCS 2798, Springer, Berlin, ISBN 3–540–20047–9; S.: 381–399

[KNS90] Wolfgang Klas, Erich J. Neuhold, Michael Schrefl: *Metaclasses in VODAK and their Application in Database Integration*; Arbeitspapiere der GMD Nr. 462, GMD – Forschungszentrum Informationstechnik GmbH, Sankt Augustin, Juli 1990, ISSN 0723–0508; 29 S.

[KOD*95] Ebru Kilic, Gökhan Ozhan, Cevdet Dengi, Nihan Kesim, Pinar Koksal, Asuman Dogaç: *Experiences in Using CORBA for a Multidatabase Implementation*; Proc. 6th Intl. Conf. on Database and Expert Systems Applications DEXA'95 (September 1995, London, UK), Workshop presentation, LNCS 978, Springer, Berlin, 1995; S. 223–230

[KSM03] Hyunchul Kang, Hosang Sung, ChanHo Moon: *Deferred Incremental Refresh of XML Materialized Views: Algorithms and Performance Evaluation;* In: Klaus-Dieter Schewe, Xiaofang Zhou (Hrsg.): Database Technologies 2003, Proc. 14th Australasian Database Conf. ADC 2003 (Februar 2003, Adelaide, Australien), Australian Computer Society, 2003, ISBN 0–909–92595–X; S. 217–226

[KuR95] Harumi A. Kuno, Elke A. Rundensteiner: *Incremental Update Propagation Algorithms for Materialized Object-Oriented Views in MultiView;* Technical Report CSE–TR–246–95, University of Michigan, Ann Arbor, 1995

[KuR96a] Harumi A. Kuno, Elke A. Rundensteiner: *The MultiView OODB View System: Design and Implementation;* Theory and Practice of Object Systems (TAPOS) 2(3): Subjectivity in Object-Oriented Systems, 1996; S. 202–225

[KuR96b] Harumi A. Kuno, Elke A. Rundensteiner: *Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views;* In: Stanley Y. W. Su (Hrsg.): Proc. 12th Intl. Conf. on Data Engineering ICDE'96 (26. Februar – 1. März 1996, New Orleans, USA), IEEE Computer Society Press, Washington, 1996, ISBN 0–8186–7240–4; S. 310–317

[LAC*93] Mary Loomis, Tom Atwood, Rick Cattell, Joshua Duhl, Guy Ferran, Drew Wade: *The ODMG Object Model;* Journal of Object-Oriented Programming, Juni 1993; S. 64–69

[LaO98] Danny B. Lange, Mitsuru Oshima: *Programming and Deploying Java Mobile Agents With Aglets;* Addison-Wesley, 1998, ISBN 0–201–32582–9

[LaR82] T. A. Landers, Ronni Rosenberg: *An Overview of MULTIBASE;* In: Hans-Jochen Schneider (Hrsg.): Distributed Data Bases, Proc. 2nd Intl. Symposium on Distributed Data Bases (1.–3. September 1982, Berlin), North-Holland, Amsterdam, 1982, ISBN 0-444-86474-1; S. 153–184

[LAZN89] Witold Litwin, Abdelaziz Abdellatif, A. Zeroual, B. Nicolas: *MSQL: A Multidatabase Language;* Information Sciences 49(1–3), 1989; S. 59–101

[LeF04] Patrick Lehti, Peter Fankhauser: *XML Data Integration with OWL: Experiences and Challenges;* In: Proc. 2004 Intl. Symp. on Applications and the Internet SAINT 2004 (26.–30. Januar 2004, Tokyo, Japan)

[Leg96] Régis Legoff: *ODMG Library;* Technical Note IRO/NOT/EDS/WP3-V2.3/ RL960820, IRO–DB Project P8629, ESPRIT–III Programme, August 1996

[LiD00] Hartmut Liefke, Susan B. Davidson: *View Maintenance for Hierarchical Semistructured Data;* In: Yahiko Kambayashi, Mukesh K. Mohania, A Min Tjoa (Hrsg.): Proc. 2nd Intl. Conf. on Data Warehousing and Knowledge Discovery DaWaK'2000 (4.–6. September 2000, London, UK), LNCS 1874, Springer, Berlin, 2000, ISBN 3–540–67980–4; S. 114–125

[Loc94] Harold W. Lockhart: *OSF DCE: guide to developing distributed applications*; MacGraw-Hill, New York, 1994, ISBN 0–07–911481–4

[LoS87] Peter C. Lockemann, Joachim W. Schmidt: *Datenbank-Handbuch;* Informatik Handbücher, Springer, Berlin, 1987, ISBN 3-540-10741-X

[Lyn83] Nancy A. Lynch: *Multilevel Atomicity: A new Correctness Criterion for Database Concurrency Control;* ACM Transactions on Database Systems 8(4), Dezember 1983, Association for Computing Machinery, New York; S. 484–502

[MAG*97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom: *Lore: A Database Management System for Semistructured Data;* SIGMOD Record 26(3), September 1997; S. 54–66

[MaH92] Frank Manola, Sandra Heiler: *An Approach to Interoperable Object Models;* In: M. Tamer Özsu, Umeshwar Dayal, Patrick Valduriez (Hrsg.): Distributed Object Management, Papers from the Intl. Workshop on Distributed Object Management IWDOM'92 (19.–21. August 1992, Edmonton, Kanada), Morgan Kaufmann, San Francisco, 1994, ISBN 1–55860–256–9; S. 304–308

[MaM91] Jean-Claude Mamou, Claudia Bauzer Medeiros: *Interactive Manipulation of Object-Oriented Views;* In: Proc. 7th Intl. Conf. on Data Engineering ICDE'91 (8.–12. April 1991, Kobe, Japan), IEEE Computer Society Press, Los Alamitos, CA, 1991, ISBN 0–8186–2138–9; S. 60–69

[MHG*92] Frank Manola, Sandra Heiler, Dimitrios Georgakopoulos, Mark F. Hornick, Michael L. Brodie: *Distributed Object Management;* Intl. Journal of Cooperative Information Systems IJCIS 1(1), 1992; S. 5–42

[Mon01] Richard Monson-Haefel: *Enterprise JavaBeans;* O'Reilly, Beijing [u.a.], 3. Ausgabe, September 2001, ISBN 0–596–00226–2.

[Mot96] Renate Motschnik-Pitrik: *Requirements and Comparison of View Mechanisms for Object-Oriented Databases;* Journal of Information Systems 21(3), Elsevier Science Ltd., 1996

[MRW*93] Peter Muth, Thomas C. Rakow, Gerhard Weikum, Peter Brössler, Christof Hasse: *Semantic Concurrency Control in Object-Oriented Database Systems;* Proc. 9th IEEE Intl. Conf. on Data Engineering ICDE'93 (19.–23. April 1993, Wien, Österreich), IEEE Computer Society Press, Los Alamitos, Calif., 1993

[MuS96] D. R. Musser, A. Saini: *C++ Programming with the Standard Template Library*; Addison-Wesley, Reading, 1996

[NEL86] Shamkant B. Navathe, Ramez Elmasri, James A. Larson: *Integrating User Views in Database Design;* IEEE Computer 19(1), Januar 1986; S. 50–62

[NeS88] Erich J. Neuhold, Michael Schrefl: *Dynamic Derivation of Personalized Views;* In: François Bancilhon, David J. DeWitt (Hrsg.): Proc. 14th Intl. Conf. on Very Large Data Bases VLDB'88 (29. August – 1. September 1988, Los Angeles, USA), Morgan Kaufmann, ISBN 0–934613–75–3; S. 183–194

[NgR89] Gio Toan Nguyen, Dominique Rieu: *Schema Evolution in Object-Oriented Database Systems;* Data and Knowledge Engineering 4(1), Juli 1989; S. 43–67

[NKOD96] Sena Nural, Pinar Koksal, Fatma Ozcan, Asuman Dogaç: *Query Decomposition and Processing in Multidatabase Systems;* Proc. Intl. Conf. on Engineering Systems Design and Analysis ESDA'96 (Juli 1996, Montpellier, Frankreich), 1996

[ODI95] Object Design, Inc.: *ObjectStore C++ API User Guide;* Release 4.0, June 1995, Object Design, Inc., Twenty Five Mall Road, Burlington, Massachusetts 01803-4194

[OHE96] Robert Orfali, Dan Harkey, Jeri Edwards: *The Essential Distributed Objects Survival Guide*; Wiley, New York, 1996, ISBN 0–471–12993–3

[OMG91] Object Management Group: *The Common Object Request Broker: Architecture and Specification;* OMG Document Number 91.12.1, Revision 1.1, Dezember 1991

[OMG95a] Object Management Group: *The Object Management Architecture Guide;* 1990, überarbeitete Ausgabe 1995

[OMG95b] Object Management Group: *CORBA Facilities Architecture Specification*; OMG Document Number 97–06–15, Revision 4.0, November 1995

[OMG97] Object Management Group: *CORBAservices: Common Object Services Specification*; OMG Document Number 97–02–24, 1995, überarbeitete Ausgabe Februar 1997

[OMG99] Object Management Group: *The Common Object Request Broker: Architecture and Specification;* OMG Document Number 99–10–07, Revision 2.3.1, überarbeitete Ausgabe Oktober 1999

[ONK*96] Fatma Ozcan, Sena Nural, Pinar Koksal, Cem Evrendilek, Asuman Dogaç: *Dynamic Query Optimization on a Distributed Object Management Platform;* Proc. 5th Intl. Conf. on Information and Knowledge Management CIKM'96 (12.–16. November 1996, Maryland, USA); S. 117–124

[ONT94a] ONTOS, Inc.: *ONTOS DB 3.0 Developer's Guide*; ONT-30-SUN-DG-1.0, May 1994, Ontos, Inc., Three Burlington Woods, Burlington, Massachusetts 01803

[ONT94b] ONTOS, Inc.: *ONTOS DB 3.0 Reference Manual, Volume 3: Function Library*; ONT-30-SUN-RMV3-1.0, May 1994, Ontos, Inc., Three Burlington Woods, Burlington, Massachusetts 01803

[ONT94c] ONTOS, Inc.: *ONTOS DB 3.0 Object SQL Guide;* ONT–30–SUN–OSQLG–1.0, April 1994, ONTOS, Inc., Three Burlington Woods, Burlington, Massachusetts 01803

[Orf99] Robert Orfali: *Client/Server Programming With Javabeans;* John Wiley, New York, 1997/1999, ISBN 0–471–18931–6

[PaS00] Christine Parent, Stefano Spaccapietra: *Database Integration: The Key to Data Interoperability;* In: Mike P. Papazoglou, Stefano Spaccapietra, Zahir Tari (Hrsg.): Advances in Object-Oriented Data Modeling, MIT Press 2000, ISBN 0-262-16189-3; S. 221–253

[PeK95] Zhiyong Peng, Yahiko Kambayashi: *Deputy Mechanisms for Object-Oriented Databases;* In: Philip S. Yu, Arbee L. P. Chen (Hrsg.): Proc. 11th Intl. Conf. on Data Engineering ICDE'95 (6.–10. März 1995, Taipei, Taiwan); IEEE Computer Society, 1995, ISBN 0–8186–6910–1; S. 333–340

[PGW95] Yannis Papakonstantinou, Hector Garcia-Molina, Jennifer Widom: *Object Exchange Across Heterogeneous Information Sources;* In: Philip S. Yu, Arbee L. P. Chen (Hrsg.): Proc. 11th Intl. Conf. on Data Engineering ICDE'95 (6.–10. März 1995, Taipei, Taiwan), IEEE Computer Society, 1995, ISBN 0–8186–6910–1; S. 251–260

[Pie99] Claudia Piemont: *Komponenten in Java: Einsatz und Entwicklung von JavaBeans mit VisualAge for Java;* dpunkt, Heidelberg, 1999, ISBN 3–932588–21–5

[POE99] Poet Software: *POET 4.0 C++ Programmer's Guide*; POET Software, On-line-Version von http://www.poet.com/programmer/index.htm

[QiW91] Xiaolei Qian, Gio Wiederhold: *Incremental Recomputation of Active Relational Expressions;* IEEE Transactions on Knowledge and Data Engineering 3(3), September 1991, IEEE, Institute of Electrical and Electronics Engineers / Computer Society, New York, NY, ISSN 1041–4347; S. 337-341

[Rad96] Elke Radeke: *Extending ODMG for Federated Database Systems;* In: Roland R. Wagner, Helmut Thoma (Hrsg.): Proc. 7th Intl. Workshop on Database and Expert Systems Applications, DEXA '96 (9.–10. September 1996, Zürich, Schweiz), IEEE Computer Society Press, Los Alamitos, 1996, ISBN 0–8186–7662–0; S. 304–312

[Rak97] Thomas C. Rakow: *Semantische Concurrency Control in objektorientierten Datenbanksystemen;* Oldenbourg, München, 1997, GMD–Bericht Nr. 284, Dissertation an der Technischen Hochschule Darmstadt, 1996, ISBN 3–486–24469–8

[RaR95a] Young-Gook Ra, Elke A. Rundensteiner: *A Transparent Object-Oriented Schema Change Approach Using View Evolution;* In: Philip S. Yu, Arbee L. P. Chen (Hrsg.): Proc. 11th Intl. Conf. on Data Engineering ICDE'95 (6.–10. März 1995, Taipei, Taiwan), IEEE Computer Society, 1995, ISBN 0–8186–6910–1; S. 165–172

[RaR95b] Young-Gook Ra, Elke A. Rundensteiner: *Towards Supporting Hard Schema Changes in TSE;* Proc. 1995 Intl. Conf. on Information and Knowledge Management CIKM'95 (28. November – 2. Dezember 1995, Baltimore, USA), ACM 1995; S. 290–295

[RaS95] Elke Radeke, Marc H. Scholl: *Functionality for Object Migration Among Distributed, Heterogeneous, Autonomous DBs;* In: Omran A. Bukhres, M. Tamer Özsu, Ming-Chien Shan (Hrsg.): Proc. 5th Intl. Workshop on Research Issues in Data Engineering – Distributed Object Management RIDE–DOM'95 (6.–7. März 1995, Taipei, Taiwan), IEEE Computer Society Press, Los Alamitos, 1995, ISBN 0–8186–7056–8; S. 58–66

[RBPF98] Antonis Ramfos, Ralph Busse, Nikos Platis, Peter Fankhauser: *A CORBA-Based Data Integration Framework*; In: Tamer M. Özsu, Asuman Dogaç, Özgür Ulusoy (Hrsg.): Proc. 3rd Biennial World Conf. on Integrated Design and Process Technology (IDPT), Workshop on Issues and Applications of Database Technology IADT'98 (5.–9. Juli 1998, Berlin), ISBN 1090–9389; Band 2, S. 176–183

[RBPF99] Antonis Ramfos, Ralph Busse, Nikos Platis, Peter Fankhauser: *An Integration Framework for CORBA Objects*; Journal of Integrated Design and Process Science (Transactions of the SDPS) 3(1), März 1999, Society for Design and Process Science (SDPS), Austin, Texas, ISSN: 1092–0617; S. 27–41

[RBP*91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design;* Prentice Hall, Englewood Cliffs, NJ, 1991

[RCDF03] Rajagopal Rajugan, Elizabeth Chang, Tharam S. Dillon, Ling Feng: *XML Views: Part 1;* In: Vladimír Maník, Werner Retschitzegger, Olga Stepánková (Hrsg.): Proc. 14th Intl. Conf. on Database and Expert Systems Applications DEXA 2003 (1.–5. September 2003, Prag, Czechische Republik), LNCS 2736, Springer, Berlin, 2003, ISBN 3–540–40806–1; S. 148–159

[RFFS95] Antonis Ramfos, Jérôme Fessy, Béatrice Finance, Véronique Smahi: *IRO–DB: A Solution for Computer Integrated Manufacturing Applications*; In: Steve Laufmann, Stefano Spaccapietra, Toshio Yokoi (Hrsg.): Proc. 3rd Intl. Conf. on Cooperative Information Systems CoopIS'95 (9.–12. Mai 1995, Schloß Wilhelminenburg Hotel, Wien, Österreich), 1995; S. 125–136

[Rou91] Nicholas Roussopoulos: *An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis;* ACM Transactions on Database Systems 16(3), September 1991; S. 535–563

[Run92] Elke A. Rundensteiner: *Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases;* In: Li-Yan Yuan (Hrsg.): 18th Intl. Conf. on Very Large Data Bases VLDB'92 (23.–27. August 1992, Vancouver, Kanada), Morgan Kaufmann, San Mateo, 1992, ISBN 1–55860–151–1; S. 187–198

[Run92b] Elke A. Rundensteiner: *A Class Integration Algorithm and its Application for Supporting Consistent Object Views;* University of California, Irvine, Technical Report #92–50, Mai 1992

[Run93] Elke A. Rundensteiner: *Tools for View Generation in Object-Oriented Databases;* In: Bharat K. Bhargava, Timothy W. Finin, Yelena Yesha (Hrsg.): Proc. 2nd Intl. Conf. on Information and Knowledge Management CIKM'93 (1.–5. November 1993, Washington DC, USA), ACM 1993, ISBN 0–89791–626–3; S. 635–644

[Run96] Elke A. Rundensteiner: *Objected-Oriented View Technology: Challenges and Promises;* In: Yahiko Kambayashi, Kazumasa Yokota (Hrsg.): Proc. Intl. Symp. on Cooperative Database Systems for Advanced Applications CODAS'96 (5.–7. Dezember 1996, Kyoto, Japan), World Scientific, 1996; S. 299–308 (bzw. S. 264–273 in Band 1 der Konferenzausgabe)

[SAD94] Cássio Souza dos Santos, Serge Abiteboul, Claude Delobel: *Virtual Schemas and Bases;* In: Matthias Jarke, Janis A. Bubenko Jr., Keith G. Jeffery (Hrsg.): Advances in Database Technology, Proc. 4th Intl. Conf. on Extending Database Technology EDBT'94 (28.–31. März 1994, Cambridge, UK), LNCS 779, Springer, Berlin, 1994, ISBN 3–540–57818–8; S. 81–94

[SAD*95] Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, William Kent: *Pegasus: A Heterogeneous Information Management System;* In: Won Kim (Hrsg.): Modern Database Systems: The Object Model, Interoperability, and Beyond; ACM Press und Addison-Wesley, 1995, ISBN 0–201–59098–0; S. 664–682

[SaH96] Stefan Sander, Willi Hasselbring: *Realisierung eines föderierten Datenbanksystems auf Basis der Standards CORBA und ODMG–93;* In: Willi Hasselbring (Hrsg.): Kurzfassungen zum 2. Workshop „Föderierte Datenbanken" (12.–13. Dezember 1996, Dortmund), Software-Technik Memo Nr. 90, Informatik 10, Universität Dortmund, 1996; S. 45–50

[SBH02] Nitesh Shrestha, Ralph Busse, Gerald Huck: *Example-Based Wrapper Generation;* In: Behrouz Homayoum (Hrsg.): Workshop Proc. 1st Eurasian Conf. on Advances in Information and Communication Technology EurAsia–ICT 2002 (29.–31. Oktober 2002, Shiraz, Iran), Books@ocg.at #161, Österreichische Computer-Gesellschaft, Wien, 2002, ISBN 3–85403–161–0; 541 S.

[Sch93] Bernhard Schiefer: *Supporting Integration & Evolution with Object-Oriented Views;* Technical Report FZI.063.3 (15/93), Forschungszentrum Informatik (FZI), Haid-und-Neu-Straße 10–14, D–76131 Karlsruhe, Juli 1993, 18 S.

[ScS90] Marc H. Scholl, Hans-Jörg Schek: *A Relational Object Model;* In: Serge Abiteboul, Paris C. Kanellakis (Hrsg.): Proc. 3rd Intl. Conf. on Database Theory ICDT'90 (12.–14. Dezember 1990, Paris, Frankreich), LNCS 470, Springer, Berlin, 1990, ISBN 3–540–53507–1; S. 89–105

[ScS91] Marc H. Scholl, Hans-Jörg Schek: *Supporting Views in Object-Oriented Databases;* IEEE Data Engineering Bulletin 14(2), Juni 1991, Special Issue on Foundations of Object-Oriented Database Systems; S. 43–47

[ScS92] Marc H. Scholl, Hans-Jörg Schek: *Survey of the COCOON Project;* In: Rudolf Bayer, Theo Härder, Peter C. Lockemann (Hrsg.): Kolloquium Objektbanken für Experten (12.–13. Oktober 1992, Stuttgart), Informatik Aktuell, Springer, Berlin, 1992, ISBN 3–540–56074–2; S. 243–253

[SDC91] Servio Logic Development Corporation: *GemStone Product Overview;* 1991

[Ses97] Roger Sessions: *COM and DCOM: Microsoft's Vision for Distributed Objects*; Wiley, New York, NY, 1997, ISBN 0–471–19381–X

[Shi81] David W. Shipman: *The Functional Data Model and the Data Language DAPLEX;* ACM Transactions on Database Systems TODS 6(1), März 1981; S. 140–173

[ShL90] Amit P. Sheth, James A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases;* ACM Computing Surveys 22(3), September 1990; S. 183–236

[ShS89] John J. Shilling, Peter F. Sweeney: *Three Steps to Views: Extending the Object-Oriented Paradigm;* In: Norman K. Meyrowitz (Hrsg.): Proc. Conf. on Object-Oriented Programming: Systems, Languages, and Applications OOPSLA'89 (1.–6. Oktober 1989, New Orleans, USA), SIGPLAN Notices 24(10), Oktober 1989; S. 353–361

[ShZ89] Gail M. Shaw, Stanley B. Zdonik: *An Object-Oriented Query Algebra;* In: Richard Hull, Ronald Morrison, David W. Stemple (Hrsg.): Proc. 2nd Intl. Workshop on Database Programming Languages DBPL–2 (4.–8. Juni 1989, Gleneden Beach, USA), Morgan Kaufmann, San Francisco, 1990, ISBN 1–55860–072–8; S. 103–112

[ShZ90] Gail M. Shaw, Stanley B. Zdonik: *A Query Algebra for Object-Oriented Databases*; Proc. 6th Intl. Conf. on Data Engineering ICDE'90 (5.-9. Februar 1990, Los Angeles, USA), IEEE Computer Society Press, Los Alamitos, CA, 1990, ISBN 0–8186–2025–0; S. 154–162

[SiK80] Abraham Silberschatz, Zvi Kedem: *Consistency in Hierarchical Database Systems;* Journal of the ACM 27(1), Januar 1980, Association for Computing Machinery, New York, NY; S. 72–80

[SJGP90] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, Spyros Potamianos: *On Rules, Procedures, Caching and Views in Data Base Systems;* In: Hector Garcia-Molina, H. V. Jagadish (Hrsg.): Proc. ACM SIGMOD Intl. Conf. on the Management of Data SIGMOD'90 (23.–25. Mai 1990, Atlantic City, USA), SIGMOD Record 19(2), Juni 1990, ACM Press, New York, N.Y., ISBN 0–89791–365–5; S. 281–290

[SLT91] Marc H. Scholl, Christian Laasch, Markus Tresch: *Updatable Views in Object-Oriented Databases;* In: Claude Delobel, Michael Kifer, Yoshifumi Masunaga (Hrsg.): Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases DOOD'91 (16.–18. Dezember 1991, München), LNCS 566, Springer, Berlin, 1991, ISBN 3–540–55015–1 und 0–387–55015–1; S. 189–207

[SpP94] Stefano Spaccapietra, Christine Parent: *View Integration: A Step Forward in Solving Structural Conflicts;* (Oktober 1990, LBD, DI, EPF Lausanne), IEEE Transactions on Knowledge and Data Engineering 6(2), April 1994; S. 258–274

[SST92] Marc H. Scholl, Hans-Jörg Schek, Markus Tresch: *Object Algebra and Views for Multi-Objectbases;* In: M. Tamer Özsu, Umeshwar Dayal, Patrick Valduriez (Hrsg.): Distributed Object Management, Papers from the Intl. Workshop on Distributed Object Management IWDOM (19.–21. August 1992, Edmonton, Kanada), Morgan Kaufmann, San Mateo, CA, 1994, ISBN 1–55860–256–9; S. 353–374

[StF02] Andrei Stoica, Csilla Farkas: *Secure XML Views;* In: Ehud Gudes, Sujeet Shenoi (Hrsg.): Research Directions in Data and Applications Security, IFIP WG 11.3 Proc. 16th Intl. Conf. on Data and Applications Security DBSec 2002 (28.–31. Juli 2002, Kings College, Cambridge, UK), IFIP Conference Proceedings 256, Kluwer, 2003, ISBN 1–4020–7541–3;. S. 133–146

[Str00] Bjarne Stroustrup: *The C++ Programming Language;* Special Edition, Addison-Wesley, Reading, Mass., 2000, ISBN 0–201–70073–5

[Sun95] SunSoft, Inc.: *Rogue Wave Tools.h++ Class Library, Introduction and Reference Manual*; SunSoft, Inc., Mountain View, USA, 1995

[TBC*87] Marjorie Templeton, David Brill, Arbee L. P. Chen, Son K. Dao, Eric Lund, Robert McGregor, Patricia Ward: *Mermaid: A Front-end to Distributed Heterogeneous Databases;* Proc. of the IEEE, Special Issue on Distributed Database Systems 75(5), Mai 1987; S. 695–708

[TeW97] Thomas Tesch, Jürgen Wäsch: *Global Nested Transaction Management for ODMG-Compliant Multi-Database Systems*; Arbeitspapiere der GMD, Technical Report 1071, GMD – Forschungszentrum Informationstechnik GmbH, St. Augustin, Mai 1997

[TeW97a] Thomas Tesch, Jürgen Wäsch: *Global Nested Transaction Management for ODMG-Compliant Multi-Database Systems*; Proc. 6th Intl. Conf. on Information and Knowledge Management CIKM'97 (10.–14. November 1997, Las Vegas, USA), (überarbeitete Version von [TeW97])

[Tre91] Markus Tresch: *A Framework for Schema Evolution by Meta Object Manipulation;* In: Jutta Göers, Andreas Heuer, Gunter Saake (Hrsg.): Proc. 3rd Workshop on Foundations of Models and Languages for Data and Objects FMLDO'91 (23.–27. September 1991, Aigen, Österreich), Informatik-Bericht 91/3, Technische Universität Clausthal, Institut für Informatik, 1991; S. 1–13

[TYI88] Katsumi Tanaka, Masatoshi Yoshikawa, Kozo Ishihara: *Schema Virtualization in Object-Oriented Databases;* Proc. 4th IEEE Intl. Conf. on Data Engineering ICDE'88 (1.–5. Februar 1988, Los Angeles, USA), IEEE Computer Society, ISBN 0–8186–0827–7; S. 23–30

[Vos87] Gottfried Vossen: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme;* Addison-Wesley, Bonn, 1987, ISBN 3-925118-64-0

[Wei86] Gerhard Weikum: *A Theoretical Foundation of Multi-Level Concurrency Control;* Proc. 5th ACM Symposium on Principles of Database Systems PODS'86 (24.–26. März 1986, Cambridge, USA), ACM Press, New York, 1986, ISBN 0–89791–179–2; S. 31–42

[WiF90] Jennifer Widom, Sheldon J. Finkelstein: *Set-Oriented Production Rules in Relational Database Systems;* In: Hector Garcia-Molina, H. V. Jagadish (Hrsg.): Proc. ACM SIGMOD Intl. Conf. on the Management of Data SIGMOD'90 (23.–25. Mai 1990, Atlantic City, USA), SIGMOD Record 19(2), Juni 1990, ACM Press, New York, N.Y., ISBN 0–89791–365–5; S. 259–270

[Wu96a] Xuequn Wu: *An Architecture for Interoperation of Distributed Heterogeneous Database Systems;* In: Roland Wagner, Helmut Thoma (Hrsg.): Proc. 7th Intl. Conf. on Database and Expert Systems Applications DEXA'96 (9.–13. September 1996, Zürich, Schweiz), LNCS 1134, Springer, Berlin, 1996, ISBN 3–540–61656–X; S. 688–697

[Wu96b] Xuequn Wu: *An Architectural Framework for Interoperation of Distributed Heterogeneous Database Systems;* Proc. 2nd IEEE Intl. Conf. on Engineering of Complex Computer Systems (21.–25. Oktober 1996, Montreal, Kanada), IEEE Computer Society Press, Los Alamitos, 1996; S. 391–394

[YoC93] Suk I. Yoo, Hai Jin Chang: *An Object-Oriented Query Model Supporting Views*; In: Shojiro Nishio, Akinori Yonezawa (Hrsg.): Object Technologies for Advanced Software, Proc. 1st JSSST Intl. Symp. ISOTAS'93 (4.–6. November 1993, Kanazawa, Japan), LNCS 742, Springer, Berlin, 1993, ISBN 3–540–57342–9; S. 251–263

[ZAB*84] Carlo Zaniolo, Hassan Aït-Kaci, David Beech, Stephanie J. Cammarata, David Maier: *Object Oriented Database Systems and Knowledge Systems;* In: Larry Kerschberg (Hrsg.): *Proc. 1. Intl. Workshop on Expert Database Systems (24.–27. Oktober 1984, Kiawah Island, USA);* ISBN 0–8053–3270–7, 1984; S. 49–65

[ZdM91] Stanley B. Zdonik, Gail Mitchell: *ENCORE: An Object-Oriented Approach to Database Modelling and Querying;* IEEE Data Engineering Bulletin 14(2), Juni 1991; S. 53–57

[Zdo93] Stanley B. Zdonik: *What Makes Object-Oriented Database Management Systems Different?;* In: Asuman Dogaç, M. Tamer Özsu, Alexandros Biliris, Timos K. Sellis (Hrsg.): Advances in Object-Oriented Database Systems, Proc. NATO Advanced Study Institute on Object-Oriented Database Systems (6.–16. August 1993, Izmir, Türkei), NATO ASI Series F: Computing and Systems Sciences Vol. 130, Springer, Berlin, 1994, ISBN 3–540–57825–0; S. 3–26

[ZhG98] Yue Zhuge, Hector Garcia-Molina: *Graph Structured Views and Their Incremental Maintenance;* Proc. 14th Intl. Conf. on Data Engineering ICDE'98 (23.–27. Februar 1998, Orlando, USA), IEEE Computer Society, 1998, ISBN 0–8186–8289–2; S. 116–125

IRO–DB Projektberichte (Deliverables)

Die Projektberichte sind im Rahmen des ESPRIT-III-Projekts IRO–DB entstanden und können unter der Adresse [W2] im WWW gefunden werden.

[D2–1/1] Georges Attali: *Specification of Local Database Adapters (D2–1/1);* Technical Report IRO/SPEC/GRAP/D2–1.1–V1.1/GA950823, IRO–DB Project P8629, ESPRIT–III Programme, August 1995

[D2–2/1] Georges Attali: *Software Delivery Document of Local Database Adapter on MATISSE (D2–2/1);* Technical Report IRO/PROT/EURI/D2-2.1-V1.0/GA951016, IRO–DB Project P8629, ESPRIT–III Programme, Oktober 1995

[D2–3/1] José-Miguel Casares: *Software Delivery Document of Local Database Adapter on Ingres (D2–3/1);* Technical Report IRO/PROT/IBER/D2-3.1-V1.0, IRO–DB Project P8629, ESPRIT–III Programme, März 1997

[D2–4/1] Christoph Mathis: *Software Delivery Document of Local Database Adapter on Ontos (D2–4/1);* Technical Report IRO/PROT/GOPS/D2–4/1–V1.0/CM, IRO–DB Project P8629, ESPRIT–III Programme, März 1997

[D2–5/1] Fei Sha: *Software Delivery Document of Local Database Adapter on O2 (D2–5/1);* Technical Report IRO/PROT/EDS/D2–5/1–V1.0/SF970228, IRO–DB Project P8629, ESPRIT–III Programme, Februar 1997

[D2–6/1] Georges Attali: *Extented Specification of Local Database Adapters (D2–6/1);* Technical Report IRO/SPEC/EURI/D2–6.1–V1.2/GA960112, IRO–DB Project P8629, ESPRIT–III Programme, Januar 1996

[D4–2/1] Peter Fankhauser, Ralph Busse, Gerald Huck: *IOM Design Specification Document (D4–2/1)*; Technical Report IRO/SPEC/GMD/D4–2.1–V1.5/PF940805, IRO–DB Project P8629, ESPRIT–III Programme, August 1994

[D4–2/3] Ralph Busse: *Full IOM Prototype (D4–2/3)*; Technical Report IRO/PROT/GMD/D4–2.3–V1.0/RB970130, IRO–DB Project P8629, ESPRIT–III Programme, Januar 1997

[D4–3/1] Ralph Busse, Georges Attali, Peter Fankhauser: *Design Document for Data Dictionary (D4–3/1)*; Technical Report IRO/SPEC/GMD/D4–3.1–V1.3/RB940722, IRO–DB Project P8629, ESPRIT–III Programme, Juli 1994

[D4–4/1] Peter Fankhauser, Regina Motz, Gerald Huck: *SIM Schema Integration Methodology (D4–4/1)*; Technical Report IRO/SPEC/GMD/D4–4.1–V1.0/ PF950210, IRO–DB Project P8629, ESPRIT–III Programme, Februar 1995

[D4–4/4] Gerald Huck: *Documentation for Implementation of Extensions to Schema Editor (D4–4/4)*; Technical Report IRO/PROT/GMD/D4–4.4–V1.0/GH951016, IRO–DB Project P8629, ESPRIT–III Programme, Oktober 1995

[D4–6/1] Thomas Tesch, Jürgen Wäsch, Lu Wenjin: *Design Specification for the Global Transaction Management (D4–6/1)*; Technical Report IRO/SPEC/GMD/D4–6.1–V1.1/TT951019, IRO–DB Project P8629, ESPRIT–III Programme, Oktober 1995

[D4–6/2] Ralph Busse: *Prototype Global Transaction Management (D4–6/2)*; Technical Report IRO/PROT/GMD/D4–6.2–V1.1/RB970205, IRO–DB Project P8629, ESPRIT–III Programme, Februar 1997

[D5–1/2] Antonis Ramfos, Ralph Busse: *Concept Demonstrator (D5–1/2);* Technical Report IRO/PROT/INTR/D5–1.2–V1.3/AR951023, IRO–DB Project P8629, ESPRIT–III Programme, Oktober 1995

Anhang A: WWW-Adressen

Die folgenden URLs führen zu öffentlich zugänglichen Seiten im World Wide Web. Sie bieten Informationen zu Projekten und Konsortien und sind teilweise bei der Erstellung der Arbeit verwendet worden. Die Adressen waren zum Zeitpunkt der Erstellung dieser Arbeit gültig. Es kann keine Gewähr übernommen werden, daß dies auch weiterhin der Fall ist.

[W1] FhG–IPSI, Fraunhofer Institut Integrierte Publikations- und Informationssysteme
<http://ipsi.fraunhofer.de/>
Bereich OASYS, Offene Adaptive Informationsmanagement-Systeme
<http://ipsi.fraunhofer.de/oasys/>

[W2] IRO-DB <http://ipsi.fraunhofer.de/oasys/projects/irodb/>

[W3] Integrators' Workbench <http://ipsi.fraunhofer.de/oasys/projects/iwb/>

[W4] VODAK <http://ipsi.fraunhofer.de/oasys/projects/vodak/>

[W5] OMG Home <http://www.omg.org/>

[W6] What's Coming in CORBA 3
<http://www.omg.org/news/pr98/compnent.html>
<http://www.omg.org/technology/corba/corba3releaseinfo.htm>

[W7] ODMG Home <http://www.odmg.org/>

[W8] ODMG-Grammatik <http://www.odmg.org/odmg93/updates/odl.html>

[W9] What's New VOL II – Februar 1999 <http://www.odmg.org/pressroom/what'snew/whatsnew.html>

[W10] What's New – Oktober 1999 <http://www.odmg.org/pressroom/what'snew/whatsnew.html>

[W11] org.odmg Java Package <http://www.odmg.org/files/odmg30.zip>

[W12] Using XML as an Object Interchange Format <http://www.odmg.org/library/readingroom/oifml.pdf>

[W13] ibermática, Spanien <http://www.ibermatica.es>

[W14] CETUS-Links for Object-Relational Mapping <http://www.cetus–links.de/oo_db_systems_3.html>

[W15] Microsoft: „DCOM Architecture"
<http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm>

[W16] Software AG: „EntireX DCOM"
<http://www.sagasoftware.com/site/solution/som/entirex/e_facts.htm>
<http://www.softwareag.com/entirex/technical/description.htm#EntireX DCOM>

[W17] Open Software Foundation (OSF) / OpenGroup <http://www.opengroup.org>

[W18] Distributed Computing Environment (DCE) <http://www.opengroup.org/dce/>

[W19] Imran Sayeed: „A Corba, DCOM primer",
in: ADTMAG — Application Development Trends, Ausgabe Juni 1998
<http://www.adtmag.com/pub/jun98/fe601.htm>

[W20] IBM Aglet Technology <http://www.trl.ibm.co.jp/aglets/>

[W21] XML Standardization Homepage <http://www.w3.org/XML>

[W22] Dublin Core Metadata Initiative <http://www.dublincore.org/>

[W23] XML/EDI Group <http://www.xmledi-group.org/>

[W24] Ronald Bourret: XML Database Products <http://www.rpbourret.com/xml/XMLDatabaseProds.htm>

[W25] XQuery 1.0: An XML Query Language, W3C Working Draft, 15. November 2002
<http://www.w3.org/TR/xquery/>

[W26] OWL Web Ontology Language Reference, W3C Working Draft, März 2003
<http://www.w3.org/TR/owl–ref/>

Anhang B: Glossar

Eine kleine Übersicht wichtiger Begriffe und Abkürzungen aus der Arbeit.

ANSI	American National Standardization Institute
Applet	Eine Javaklasse, die im Rahmen des WWW an den Client übertragen werden kann und dann direkt im Browser läuft.
CAF	Composed Access Facility, Laufzeitumgebung der CORBA-Sichtabbildung
CLI	Call Level Interface (ANSI)
Client	Datennutzer in einer „Client-Server-Verbindung“
CORBA	Common Object Request Broker Architecture (OMG)
COSS	CORBA Object Services Specification (OMG)
DCOM	Distributed Common Object Model (Microsoft)
GOID	Global OID (IRO–DB)
HTTP	HyperText Transfer Protocol; das im WWW verwendete Datenaustausch-protokoll.
IDD	Interoperable Data Dictionary (IRO–DB)
IDL	Interface Definition Language (OMG)
Importklasse	Eine virtuelle Klasse für den Zugriff auf externe Objekte.
ISO	International Organization for Standardization
ISO Latin-1	Westeuropäische Zeichensatzcodierung gemäß ISO 8859–1
Kategorie	Das Sichtsystem unterscheidet zwischen normalen und virtuellen Klassen und definiert hierfür die drei Kategorien *normal*, *importierend* und *konvertierend*.
Konvertie-rungsklasse	Eine virtuelle Klasse, die ihren Extent ausschließlich aus anderen virtuellen Klassen und lokalen Datenbankklassen bestimmt. Es erfolgt kein direkter Zugriff auf externe Objekte.
OA	Object Adapter, generische Server-seitige Schnittstelle zum ORB (OMG)
ODL	Object Definition Language (ODMG)
ODL*	Um Sichtelemente erweiterte ODL-Syntax, siehe Kapitel 3.4.
OORDA	Object-Oriented Remote Data Access (siehe RDA)
OQL	Object Query Language (ODMG)
ODMG	Object Data(base) Management Group
OEM	Object Exchange Model, ein semistrukturiertes Datenmodell
OID	Objekt-Identifier
OMA	Object Management Architecture (OMG)
OMG	Object Management Group
OODB	objektorientierte Datenbank
OODBMS	objektorientiertes Datenbank-Management-System
ORB	Object Request Broker (OMG)
Query	Anfrage an ein Datenbanksystem

RDA	Remote Data Access
Repository	Metadatenspeicher bei CORBA (OMG)
RMI	Remote Method Invocation, Methodenaufruf auf externen Rechnern
ROA	Remote Object Access
Server	Datenanbieter in einer „Client-Server-Verbindung“
Skeleton	Server-seitige, schemaspezifische Schnittstelle zum ORB (OMG)
SQL	Structured Query Language (ANSI)
STL	Standard Template Library, C++-Funktionsbibliothek für strukturierte Datentypen
Stub	Client-seitige, schemaspezifische Schnittstelle zum ORB (OMG)
Virtuelle Klasse	Eine Klasse aus meinem Sichtmodell, die ihre Instanzen, Attribute und Methoden von anderen Klassen ableitet. Sie hat keinen persistenten Extent sondern berechnet ihn mit Hilfe einer Anfrage.
VML	VODAK Model Language
VQL	VODAK Query Language
W3C	World Wide Web Consortium; Standardisierungsgremium des WWW
WWW	World Wide Web, Hypertext im Internet
XML	Extensible Markup Language; Auszeichnungssprache vom W3C

Anhang C: Beispielschema

Die folgenden Klassendefinitionen werden als Beispielschema in dieser Arbeit verwendet. Bild 28 zeigt alle Klassen in der Übersicht.

Externes Schema „ADRDB“

ADRDB ist eine Personendatenbank, die für jede Person Name, Geburtsdatum und Adresse verwaltet.

```
interface Person
    (   extent persons
        key (lname,fname) ) : persistent
    {
    attribute       string              lname;
    attribute       string              fname;
    attribute       string              birthdate;
    relationship    Address             address   inverse Address::residents;
    };

interface Address
    (   extent addresses
        key () ) : persistent
    {
    attribute       string              street;
    attribute       string              city;
    relationship    set<Person>         residents inverse Person::address;
    }
```

Externes Schema „PUBDB“

Die Datenbank PUBDB beschreibt Autoren und ihre Publikationen. Für jeden Autor in PUBDB sei auch eine entsprechende Person in ADRDB vorhanden.

```
interface Author
    (   extent authors
        key (lastname, firstname) ) : persistent
    {
    attribute       string                  lastname;
    attribute       string                  firstname;
    relationship    set<Publication>        publications  inverse Publication::authors;
                    long                    pubcount ();
    };

interface Publication
    (   extent publications
        key pubid ): persistent
    {
    attribute       string              pubid;
    attribute       string              title;
    attribute       enum { book, article, journal }
                                        type;
    attribute       float               price;      // in $
    relationship    set<Author>         authors   inverse Author::publications;
    }
```

Importschema

Die folgenden Klassen importieren die externen Schemas in das Sichtsystem. Alle Klassen erhalten den ursprünglichen Datenbanknamen als Namenspräfix.

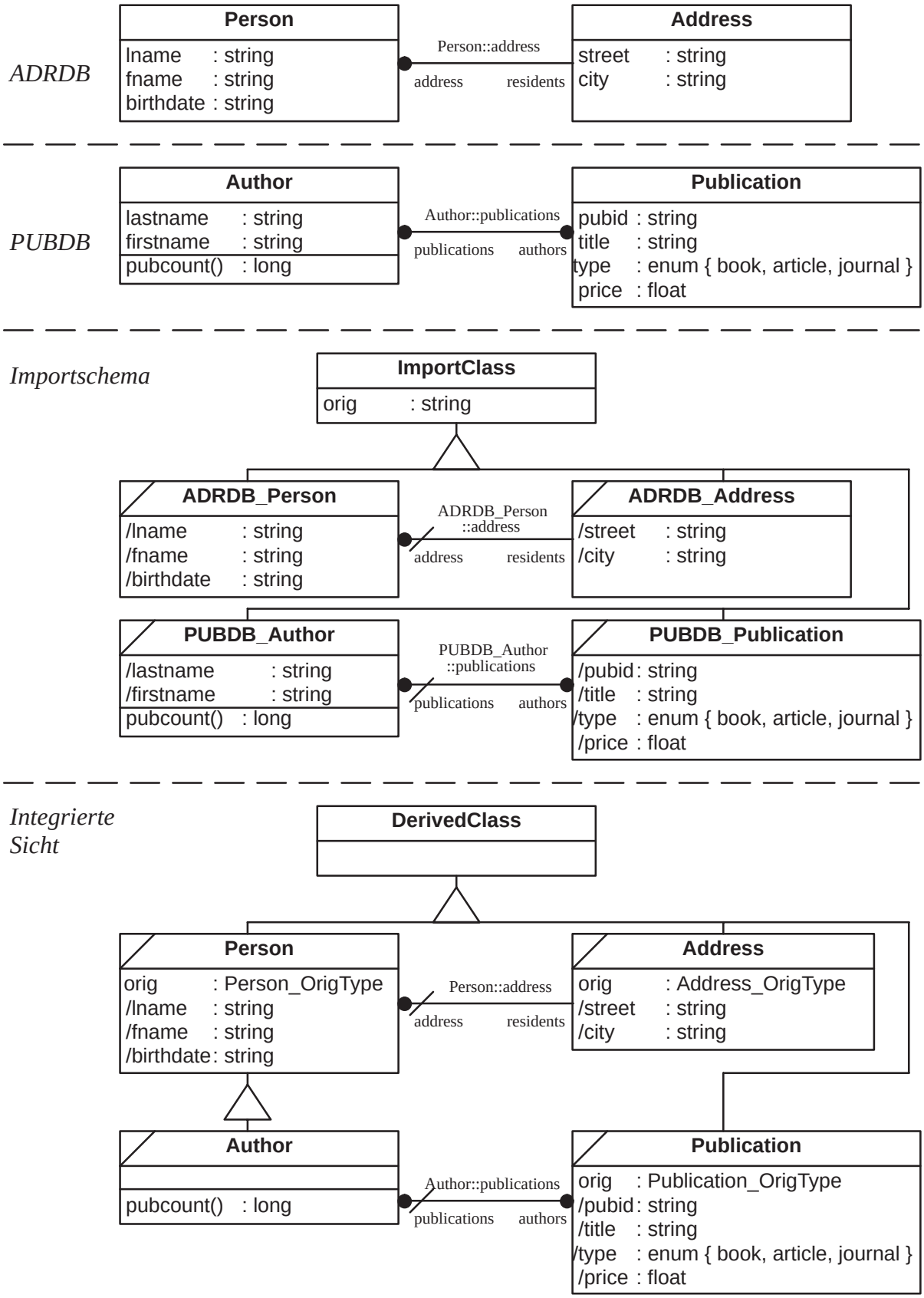


Bild 28: Beispielschema in Rumbaugh-Notation

```
interface PUBDB_Author : ImportClass
    (   extent pubdb_authors imports Author from PUBDB::authors;
        key (lastname, firstname) ) : virtual
    {
    attribute       string                          lastname;
    attribute       string                          firstname;
    relationship    set<PUBDB_Publication>          publications  inverse PUBDB_Publication::authors;
                    long                            pubcount ();
    };

interface PUBDB_Publication : ImportClass
    (   extent pubdb_publications imports Publication from PUBDB::publications;
        key pubid ): virtual
    {
    attribute       string                          pubid;
    attribute       string                          title;
    attribute       enum { book, article, journal } type;
    attribute       float                           price;     // in $
    relationship    set<PUBDB_Author>               authors    inverse PUBDB_Author::publications;
    }

interface ADRDB_Person : ImportClass
    (   extent adrdb_persons imports Person from ADRDB::persons;
        key (lname,fname) ) : virtual
    {
    attribute       string                  lname;
    attribute       string                  fname;
    attribute       string                  birthdate;
    relationship    ADRDB_Address           address   inverse ADRDB_Address::residents;
    };

interface ADRDB_Address : ImportClass
    (   extent adrdb_addresses imports Address from ADRDB::addresses;
        key (street,city) ) : virtual
    {
    attribute       string                  street;
    attribute       string                  city;
    relationship    Set<ADRDB_Person>residents inverse ADRDB_Person::address;
    }
```

Integrierte Sicht

Die integrierte Sicht vereinigt die beiden Datenbanken zu einem gemeinsamen Schema.

```
typedef struct { ADRDB_Person person; PUBDB_Author author; }   Person_OrigType;
typedef struct { ADRDB_Address addess; }                       Address_OrigType;
typedef struct { PUBDB_Publication publication; }              Publication_OrigType;

interface Person : DerivedClass
    (   extent persons is   select Person ( orig: struct( person:p, author:a ) )
                            from p in adrdb_persons, a in pubdb_authors
                            where p.lname=a.lastname and p.fname=a.firstname
                        union
                            select Person( orig: struct( person:p, author:nil ) )
                            from p in adrdb_persons
                            where not exists a in pubdb_authors: (p.lname=a.lastname and p.fname=a.firstname);
        key (lname,fname) ) : virtual
    {
    private     attribute       Person_OrigType  orig;
                attribute       string           lname       is this.orig.person.lname;
                attribute       string           fname       is this.orig.person.fname;
                attribute       string           birthdate   is this.orig.person.birthdate;
                relationship    Address          address     inverse Address::residents is
                    element( select a
                            from a in addresses
                            where a.orig.address = this.orig.person.address);
    };
```

```
interface Author : Person
    (   extent authors is   select (Author)p
                            from p in persons
                            where p.orig.author != nil;
        key (lname, fname) ) : virtual
    {
    relationship  set<Publication>   publications  inverse Publication::authors is
                                                                    select p
                                                                    from p in publications
                                                                    where p.orig.publication in this.orig.author.publications;
                  long               pubcount ();
    };

interface Address : DerivedClass
    (   extent addresses is   select Address ( orig: struct( address:a ) )
                              from a in adrdb_addresses;
        key (street,city) ) : virtual
    {
    private   attribute      Address_OrigType orig;
              attribute      string           street      is this.orig.address.street;
              attribute      string           city        is this.orig.address.city;
              relationship   set<Person>      residents   inverse Person::address is
                                                                    select p
                                                                    from p in persons
                                                                    where p.orig.person in this.orig.address.residents;
    };

interface Publication : DerivedClass
    (   extent publications is  select Publication ( orig: struct( publication:p ) )
                                from p in pubdb_publications;
        key pubid ) : virtual
    {
    private   attribute      Publication_OrigType  orig;
              attribute      string                          pubid     is this.orig.publication.pubid;
              attribute      string                          title     is this.orig.publication.title;
              attribute      enum { book, article, journal } type      is this.orig.publication.type;
              attribute      float                           price     is this.orig.publication.price * 0.9011;   // in EUR
              relationship   set<Author>                     authors   inverse Author::publications is
                                                                    select a
                                                                    from a in authors
                                                                    where a.orig.author in this.orig.publication.authors;
    };
```

Anhang D: Die ODL- und ODL*-Grammatiken

Die nachfolgende ODL-Grammatik ist die offizielle Version, die von der Object Database Management Group auf ihrer Webseite [W8] bereitgestellt wird. Sie entspricht der Version 1.1 des Standards ([Cat94]), weist aber die folgenden Unterschiede auf:[49]

- array_type erlaubt nicht mehr die Verwendung von **sequence**
- die Angabe einer Inversen in rel_dcl ist hier optional
- **bag** und **array** sind als rel_collection_type nicht erlaubt
- alle Schlüsselworte, die in den gedruckten Versionen groß oder mit großer Initiale geschrieben sind, werden hier klein geschrieben

Die Grammatik verwendet ein vereinfachtes ‘yacc’-Format, bei dem optionale Elemente durch [...] gekennzeichnet sind. Token sind in der Grammatik fett gedruckt. Eine entsprechende ‘lex’-Datei ist auf der Webseite ebenfalls zu finden.

```
model                : specification ;
specification        : definition [ specification ] ;
definition           : type_dcl ; | const_dcl ; | except_dcl ; | interface ; | module ; ;
module               : module identifier { specification } ;
interface            : interface_dcl | forward_dcl ;
interface_dcl        : interface_header [ : persistence_dcl ] { [ interface_body ] } ;
persistence_dcl      : persistent | transient ;
forward_dcl          : interface identifier ;
interface_header     : interface identifier [ : inheritance_spec ] [ type_property_list ] ;
type_property_list   : ( [ extent_spec ] [ key_spec ] ) ;
extent_spec          : extent identifier ;
key_spec             : key key_list | keys key_list ;
key_list             : key [ , key_list ] ;
key                  : property_name | ( property_list ) ;
property_list        : property_name [ , property_list ] ;
property_name        : scoped_name ;
interface_body       : export [ interface_body ] ;
export               : type_dcl ; | const_dcl ; | except_dcl ; | attr_dcl ; | rel_dcl ;| op_dcl ; ;
inheritance_spec     : scoped_name [ , inheritance_spec ] ;
scoped_name          : identifier | :: identifier | scoped_name :: identifier ;
const_dcl            : const const_type identifier = const_expr ;
const_type           : integer_type | char_type | boolean_type | floating_pt_type | string_type
                     | scoped_name ;
const_expr           : or_expr ;
or_expr              : x_or_expr | or_expr | x_or_expr ;
x_or_expr            : and_expr | x_or_expr ^ and_expr ;
and_expr             : shift_expr | and_expr & shift_expr ;
shift_expr           : add_expr | shift_expr >> add_expr | shift_expr << add_expr ;
add_expr             : mult_expr | add_expr + mult_expr | add_expr – mult_expr ;
mult_expr            : unary_expr | mult_expr * unary_expr | mult_expr / unary_expr
                     | mult_expr % unary_expr ;
unary_expr           : [ unary_operator ] primary_expr ;
unary_operator       : – | + | ~ ;
```

49. Diese Änderungen sind in der Grammatik von Version 1.2 ([Cat96]) überhaupt nicht, in Version 2.0 ([CaB97]) nur teilweise wiederzufinden. Es existiert demnach noch immer keine verläßliche Grammatik für ODL.

primary_expr : scoped_name | literal | **(** const_expr **)** ;
literal : integer_literal | string_literal | character_literal | floating_pt_literal
| boolean_literal ;
boolean_literal : **true** | **false** ;
positive_int_const : const_expr ;
type_dcl : **typedef** type_declarator | struct_type | union_type | enum_type ;
type_declarator : type_spec declarators ;
type_spec : simple_type_spec | constr_type_spec ;
simple_type_spec : base_type_spec | template_type_spec | scoped_name ;
base_type_spec : floating_pt_type | integer_type | char_type | boolean_type | octet_type
| any_type ;
template_type_spec : array_type | string_type
constr_type_spec : struct_type | union_type | enum_type ;
declarators : declarator [**,** declarators] ;
declarator : simple_declarator | complex_declarator ;
simple_declarator : identifier ;
complex_declarator : array_declarator ;
floating_pt_type : **float** | **double** ;
integer_type : signed_int | unsigned_int ;
signed_int : signed_long_int | signed_short_int ;
signed_long_int : **long** ;
signed_short_int : **short** ;
unsigned_int : unsigned_long_int | unsigned_short_int ;
unsigned_long_int : **unsigned long** ;
unsigned_short_int : **unsigend short** ;
char_type : **char** ;
boolean_type : **boolean** ;
octet_type : **octet** ;
any_type : **any** ;
struct_type : **struct** identifier **{** member_list **}** ;
member_list : member [member_list] ;
member : type_spec declarators **;** ;
union_type : **union** identifier **switch (** switch_type_spec **) {** switch_body **}** ;
switch_type_spec : integer_type | char_type | boolean_type | enum_type | scoped_name ;
switch_body : case [switch_body] ;
case : case_label_list element_spec **;** ;
case_label_list : case_label [case_label_list] ;
case_label : **case** const_expr **:**| **default :** ;
element_spec : type_spec declarator ;
enum_type : **enum** identifier **{** enumerator_list **}** ;
enumerator_list : enumerator [**,** enumerator_list] ;
enumerator : identifier ;
array_type : **array <** simple_type_spec [**,** positive_int_const] **>** ;
string_type : **string** [**<** positive_int_const **>**] ;
array_declarator : identifier array_size_list ;
array_size_list : fixed_array_size [array_size_list] ;
fixed_array_size : **[** positive_int_const **]** ;
attr_dcl : [**readonly**] **attribute** domain_type identifier [fixed_array_size] ;
domain_type : simple_type_spec | struct_type | enum_type
| attr_collection_specifier literal | attr_collection_specifier identifier ;
attr_collection_specifier : **set** | **list** | **bag** | **array** ;
rel_dcl : **relationship** target_of_path identifier [**inverse** inverse_traversal_path]
[**{ order_by** attribute_list **}**] ;

```
target_of_path          : identifier | rel_collection_type < identifier > ;
inverse_traversal_path  : identifier :: identifier ;
attribute_list          : scoped_name [ , attribute_list ] ;
rel_collection_type     : set | list
except_dcl              : exception identifier { [ member_list ] } ;
op_dcl                  : [ oneway ] op_type_spec identifier param_dcls
                            [ raises ( scoped_name_list ) ] [ context ( string_literal_list ) ] ;
op_type_spec            : simple_type_spec | void ;
param_dcls              : ( [ param_dcl_list ] ) ;
param_dcl_list          : param_dcl [ , param_dcl_list ] ;
param_dcl               : param_attribute simple_type_spec declarator ;
param_attribute         : in | out | inout ;
scoped_name_list        : scoped_name [ , scoped_name_list ] ;
string_literal_list     : string_literal [ , string_literal_list ] ;
```

Für die eingebettete Spezifikation von Sichten habe ich die folgenden Änderungen eingeführt:

```
persistence_dcl         : persistent | transient | virtual ;
extent_spec             : extent identifier [ ( derivation_clause | import_clause) ; ] ;
attr_dcl                : [ virtual | private ] [ readonly ] attribute domain_type identifier
                            [ fixed_array_size ] [ derivation_clause ];
rel_dcl                 : [ virtual ] relationship target_of_path identifier
                            [ inverse inverse_traversal_path ] [ { order_by attribute_list } ]
                            [ derivation_clause ];
import_clause           : imports external_type from external_db :: external_extent
                            [ [ identifier ] where query ] ;
derivation_clause       : is query ;
query                   : *** siehe OQL-Grammatik ***
```

Für die alternative separate Spezifkation wird die folgende Grammatik für die Mapping-Datei verwendet:

```
mapping_list            : mapping [ mapping_list ] ;
mapping                 : mapping identifier ( import_mapping | derivation_mapping ) ; ;
import_mapping          : imports external_type from external_db :: external_extent
                            [ [ identifier ] where query ] ;
derivation_mapping      : { extent_clause [ prop_clauses ] } ;
extent_clause           : def_ext query ; ;
prop_clauses            : prop_clause [ prop_clauses ] ;
prop_clause             : att_clause | rel_clause ;
att_clause              : def_att identifier as query ; ;
rel_clause              : def_rel identifier as query ; ;
query                   : *** siehe OQL-Grammatik ***
```

Anhang E: Die OQL-Grammatik

Dies ist die OQL-Grammatik zu Version 1.1 des ODMG-Standards ([Cat94]). Wie oben sind Schlüsselwörter fett gedruckt und optionale Elemente in [...] eingeschlossen.

```
query_program    : [ define_list ] query ;
define_list      : define_query [ define_list ] ;
define_query     : define identifier as query ; ;
query            : basic
                 | arithm
                 | comparison
                 | boolean_expr
                 | constructor
                 | accessor
                 | conversion
                 | set_expr
                 | coll_expr ;
basic            : nil
                 | true
                 | false
                 | integer_literal
                 | float_literal
                 | character_literal
                 | string_literal
                 | entry_name
                 | query_name
                 | bind_argument
                 | ( query ) ;
arithm           : query + query
                 | query – query
                 | query * query
                 | query / query
                 | – query
                 | query mod query
                 | abs ( query ) ;
comparison       : query = query
                 | query != query
                 | query <= query
                 | query < query
                 | query >= query
                 | query > query ;
boolean_expr     : not query
                 | query and query
                 | query or query ;
constructor      : type_name ( [ query ] )
                 | type_name ( field_list )
                 | struct ( field_list )
                 | set ( [ query_list ] )
                 | bag ( [ query_list ] )
                 | list ( [ query_list ] )
                 | array ( [ query_list ] ) ;
query_list       : query [ , query_list ] ;
field_list       : identifier: query [ , field_list ];
accessor         : query dot attribute_name
                 | query dot relationship_name
                 | query dot operation_name
                     ( [ query_list ] )
                 | query [ query ]
                 | query [ query : query ]
                 | first ( query )
                 | last ( query ) ;
dot              : . | -> ;
conversion       : listtoset ( query )
                 | element ( query )
                 | flatten ( query )
                 | ( class_name ) query ;
set_expr         : query intersect query
                 | query union query
                 | query except query ;
coll_expr        : for all identifier in query :
                     query
                 | exists identifier in query :
                     query
                 | query in query
                 | select [ distinct ] query
                     from vardecl_list
                     [ where query ]
                 | sort identifier in query
                     by query_list
                 | count ( query )
                 | sum ( query )
                 | min ( query )
                 | max ( query )
                 | avg ( query )
                 | group identifier in query
                     by ( field_list )
                     [ with ( field_list ) ] ;
vardecl_list     : identifier in query
                     [ , vardecl_list ] ;
entry_name       : identifier ;
query_name       : identifier ;
type_name        : identifier ;
class_name       : identifier ;
attribute_name   : identifier ;
relationship_name : identifier ;
operation_name   : identifier ;
```

Anhang F: Syntax der Konfigurationsdatei

Der Präprozessor benötigt eine Konfigurationsdatei, um Codeelemente bei der Sprachbindung zu berücksichtigen, die nicht aus der ODL*-Spezifikation hergeleitet werden können (siehe Kapitel 4.6.1). Die dazu verwendete Konfigurationsdatei hat das folgende Format:

```
control_file      : commands ;
commands          : command [ commands ] ;
command           : comment | prefixblock | postfixblock | classblock ;
comment           : # rest_of_line ;
prefixblock       : PREFIX [ codeblock ] . ;
postfixblock      : POSTFIX [ codeblock ] . ;
classblock        : CLASS interface_name [ codeblock ] . ;
codeblock         : line_of_text [ codeblock ] ;
```

Die Datei ist zeilenorientiert, das heißt, die Blockanfänge und der schließende Punkt stehen jeweils auf einer eigenen Zeile. Die Codezeilen des Prefixblocks werden in der Headerdatei vor der ersten Klassendefinition aber nach allen allgemeinen Definitionen eingefügt. Die Zeilen des Postfixblocks kommen an das Ende der Datei. Die Zeilen eines Klassenblocks werden in die Klassendeklaration von interface_name eingefügt. Sie werden als protected definiert und stehen direkt vor der schließenden geschweiften Klammer.

Die folgenden Zeilen geben ein Beispiel für eine solche Konfigurationsdatei. In den meisten Fällen wird die Datei aber einfach leer sein, da im allgemeinen keine Erweiterungen der Headerdatei notwendig sind.

```
#######################################################
# Automatically generated control file "exSchema.cfg"
#######################################################

PREFIX
// include a file with accounting definitions
#include "Accounting.h"
.

################# Import Class ADRDB_Person ################
CLASS ADRDB_Person
    // the following function is used for internal accounting:
    void do_accounting (Account ac);
.

################# Import Class ADRDB_Address ################
CLASS ADRDB_Address
    // the following function is used for internal accounting:
    void do_accounting (Account ac);
.

################# Derived Class Person ################
CLASS Person
    // no specific members
.

################# Derived Class Address ################
CLASS Address
    // no specific members
.

POSTFIX
// some inline functions for convenience
inline InterfaceDef* PersonMetaclass () { return Person::metaclass(); };
inline InterfaceDef* AddressMetaclass () { return Address::metaclass(); };
.
# End of file exSchema.cfg
```

Anhang G: Anmerkungen zu ODMG–93

Die Arbeit mit ODMG–93 hat sich als sehr schwierig erwiesen. Dies ist in erster Linie darauf zurückzuführen, daß sich der Standard ständig in der Entwicklung befand. Die ersten Veröffentlichungen waren zum großen Teil eine Sammlung mehr oder weniger weit fortgeschrittener Ideen, wie der Standard einmal aussehen soll. Besonders die erste Ausgabe 1.0 litt sehr darunter, daß die Kapitel von verschiedenen Autoren geschrieben worden waren. Es fehlte an einer buchübergreifenden Korrektur, um die Inkonsistenzen und Widersprüche zwischen den einzelnen Kapiteln zu entfernen. Neben diesen offensichtlichen Widersprüchen, waren wesentliche Teile auch unklar beschrieben oder fehlten völlig ([Kay94], [Kim94a], [Kim94b], [CKZ94]). Als Folge davon war man gezwungen, Annahmen über die weitere Entwicklung zu treffen und eine eigene Interpretation des Standards zu verwenden. In Release 1.1 wurden viele dieser Unstimmigkeiten behoben, aber dennoch blieben noch wesentliche Fragen unbeantwortet. Da diese fehlenden Informationen auch diese Arbeit beeinflußten, sind sie im folgenden näher aufgeführt. In Release 1.2 sind viele dieser Probleme behoben und es hat größere Design-Änderungen gegeben. Der in dieser Arbeit beschriebene Ansatz ist jedoch auf Basis von Release 1.1 implementiert worden, da Release 1.2 erst zu spät erschienen ist. Die notwendigen Anpassungen in Bezug auf die neueren ODMG-Versionen sind in Kapitel 6.2.1 beschrieben.

Neben den in den obigen Literaturstellen veröffentlichten Kritikpunkten sind besonders die folgenden konkreten Probleme beim Studium des Standards zu Tage getreten:

- Unklare Verwendung des Begriffs *object*
  Obwohl der Standard zu Beginn die Begriffe *object*, *denotable object* und *mutable object* klar definiert, wird dies innerhalb des Textes nicht eingehalten. Der Begriff *object* wird meist doch wieder unqualifiziert benutzt. In dieser Arbeit bezeichnet *object* grundsätzlich ein *mutable object* in obigem Sinne.
- Mangelnde Interpretation von strukturierten Objekten (persistenten Literalen)
  Wie bereits auf Seite 11 angemerkt, ist die Definition von strukturierten Objekten nicht nachvollziehbar. Darüber hinaus wird in der Typdefinition zwischen literalen und objektwertigen Strukturen unterschieden (`Immutable_Set<>` versus `Set<>`), während im Rest des Buchs aber immer genau die objektwertigen Varianten als Literale verwendet werden!
- Verwendung objektwertiger Attribute als unidirektionale Relationships
  Je nach Abschnitt des Buchs sind Inverse für Relationships zwingend oder nicht. Die Verwendung von objektwertigen Attributen als unidirektionale Relationships kann als Folge davon verstanden werden.
- Fehlende Orthogonalität im Typsystem
  Das Typsystem in Release 1.0 und 1.1 wies deutliche Brüche in der Orthogonalität auf. In Strukturen, Attributen und Typdefinitionen waren jeweils andere Teilmengen des Typsystems erlaubt. Dies ist in Release 1.2 wesentlich verbessert worden. Der Zusammenhang zwischen Arrays fester und variabler Größe (`int[15]` versus `Array<int>`) wird nicht geklärt.
- Vermischung von Schemaebenen
  Die Idee, Typen als *first-class objects* im Modell aufzufassen ([Cat94, S. 12]), führt dazu, daß die Typbeschreibungsebene und die konzeptuelle Ebene ständig miteinander vermischt werden. Es fällt schwer, nachzuvollziehen, wann ein Typ als Instanz des Metaschemas aufgefaßt wird und wann als strukturelle Eigenschaft anderer Objekte oder Literale. Ein weiteres Problem ist, daß genau diese Typ-als-Objekt-Eigenschaft in der Sprachbindung ebenfalls wieder fallen gelassen wird (vgl. Kapitel 4.2.4).

- Unvollständige Spezifikation der generischen Basisdatentypen
  Das von der ODMG angestrebte Ziel eines Datenaustauschstandards ([Cat94, S. 2]) bleibt zweifelhaft, solange die Genauigkeiten und Darstellungen der einzelnen Datentypen von der Sprachbindung abhängen ([Cat94, S. 20]). Wenn weder die physische Größe der Datentypen (Bitanzahl), noch ihre Speicherausrichtung (little/big endian), noch ihre Interpretation (etwa Character als ISO Latin-1 oder ASCII oder Unicode) standardisiert sind, wie soll dann der angestrebte sprachunabhängige Datenaustausch je zustande kommen? Neben diesen physischen Details sind die Sprachbindungen an sich schon so unterschiedlich, daß der angestrebte *shared object space* ([Cat94, S. 160]) illusorisch ist.

- Unvollständige Spezifikation des Namensraums
  ODMG–93 geht sehr freizügig mit Groß- und Kleinschreibung um, ohne eine genaue Interpretation zu geben. Da es auf dem IDL-Standard beruht, kann man davon ausgehen, daß es die Regel übernimmt, daß es nur einen globalen Namensraum gibt und daß Groß- und Kleinbuchstaben zwar nicht unterschieden werden, daß ein Identifier aber stets in identischer Schreibweise erscheinen muß. Unterscheiden sich zwei Identifier nur in ihrer Groß-/Kleinschreibung, so gilt dies in IDL als Syntaxfehler ([OMG99, Kap. 3]). Ob dies auch für ODMG–93 gilt, ist dem Standard nicht zu entnehmen. Das Buch selbst verwendet in jedem Kapitel andere Schreibweisen. In dieser Arbeit verwende ich die ODL-Grammatik aus Anhang D, die der auf der Webseite der ODMG verfügbaren yacc-Grammatik entspricht, sich aber von den gedruckten Versionen in [Cat94] und [Cat95] unterscheidet. Eine weitere Unklarheit sind die in einem Identifier erlaubten Zeichen. IDL erlaubt sämtliche alphanumerischen Zeichen des ISO Latin-1 Zeichensatzes sowie den Unterstrich, wobei jeder Bezeichner mit einem Buchstaben beginnen muß. ODMG–93 enthält hierzu überhaupt keine Angaben, die auf der ODMG-Webseite verfügbare Parserspezifikation für den Parsergenerator lex erlaubt aber neben Ziffern und dem Unterstrich nur die Buchstaben A bis Z. Da allerdings in der besagten Spezifikation die Forderung nach einem Buchstaben am Anfang fehlt, erscheint sie äußerst unglaubhaft, da hiernach auch reine Ziffernkombinationen gültige Bezeichner wären. Ein Problem, das allerdings auch in IDL nicht gelöst ist, ist die direkte Abbildung der Bezeichner nach C bzw. C++. Da diese Sprachen bisher nur reine ASCII-Bezeichner verwenden können, sollte die Verwendung von Sonderzeichen in Bezeichnern auf jeden Fall vermieden werden.

- Mischen von transienten und persistenten Klassen
  Laut Syntax ist die Angabe von persistent bzw. transient bei der Interfacedefinition optional; ein Default-Verhalten wird aber nicht definiert. Außerdem wird nicht festgelegt, ob man transiente Subklassen von persistenten Klassen oder persistente Subklassen von transienten Klassen anlegen darf, und wie solches zu interpretieren ist.

- Methodenbindung
  Der Standard definiert nicht, ob Methoden zur Laufzeit oder Definitionszeit gebunden werden, ob es sich also im Sinne von C++ um virtuelle Methoden handelt. Es wird auch nicht definiert, ob Methoden in einer Subklasse redefiniert werden dürfen.

- Mangelnde Typkonsistenz zwischen Modell und ODL
  Die Modelltypen Date, Time, Timestamp, Interval und Bit_String sowie alle Immutable Collections lassen sich in ODL nicht ausdrücken. Umgekehrt definiert ODL vier Ganzzahl- und zwei Gleitkommatypen, während es im Modell nur jeweils einen gibt. Der Sequence-Datentyp erscheint genauso wenig im Modell wie Arrays fester Größe (z.B. int[4]).

- Keine einheitliche ODL-Syntax
  Die verwendete Spezifikationssprache war in annähernd jedem Kapitel mehr oder weniger stark voneinander verschieden. Ich verwende hier die Syntax, die offiziell auf der ODMG-Webseite für Release 1.1 zur Verfügung gestellt worden ist (siehe Anhang D).
- Konstruktoren und Klassenmethoden
  ODL erlaubt nur die Spezifikation von Instanzmethoden. Konstruktoren zur Objekterzeugung und -intialisierung sowie Klassenmethoden fehlen völlig.
- Fehlende Updates in OQL
  OQL bietet keine Möglichkeit, Attributwerte direkt zu ändern. Alle Updates müssen über Methoden implementiert werden. Denkbare Alternativen wären, den Attributnamen als Methode aufzufassen oder eine generische „set_value“-Methode aufzurufen.
- Namenskonflikte bei multipler Vererbung
  Die im generischen Teil beschriebene explizite Auflösung von Namenskonflikten mittels „<new property> redefines <supertype>.<old_property>“ ([Cat94, S. 15]) ist in ODL nicht vorgesehen.
- Inkonsistente OML-Funktionen
  Die verschiedenen vordefinierten Operationen auf den strukturierten Datentypen unterscheiden sich in den verschiedenen Teilen des Standards.
- Defaultwerte
  Die Definition von Defaultwerten für Attribute ([Cat94, S. 22]) wird ebenfalls von ODL nicht unterstützt.
- Mangelnde Funktionalität in der C++-Sprachbindung
  Die C++-Sprachbindung unterstützt weder die get_value und set_value Operationen noch Defaultwerte für Attribute. Nullwerte werden nicht unterstützt, und das Typsystem unterscheidet sich ebenfalls in wesentlichen Teilen (siehe Kapitel 4.2.4 und Anhang H.1).

Neben diesen konkreten Arbeitsproblemen hat aber auch das Modell gewisse Einschränkungen, die sich auf die Entwicklung eines Sichtsystems auswirken: die Zugehörigkeit eines Objekts zu genau einer Klasse und die fehlende Migrationsfähigkeit schränken die Möglichkeiten für eine Sichtrealisierung, ohne das Modell zu verändern, deutlich ein. Für eine Implementierung kommt noch das Fehlen von Nullwerten und eines Metaschemas in der Sprachbindung erschwerend hinzu.

Weitere Kritikpunkte ergeben sich speziell aus der unvollständigen C++-Sprachbindung von Release 1.1. Genau wie der generische Teil des Standards ist auch die Sprachbindung teilweise problematisch und ungenau. Die Sprachbindung selbst ist nicht konkret definiert, sondern nur anhand von Beispielen erläutert. Diese Beispiele widersprechen sich wiederum in einzelnen Fällen. Die in dieser Arbeit verwendete Abbildung stellt nur eine mögliche Interpretation dar. Andere C++-Sprachbindungen können davon abweichen. Die wesentlichsten Probleme sind:

- oql-Funktion überfordert den Compiler
  Bei unserer Projektarbeit konnte nur eine Teilmenge der im Standard definierten oql-Funktionen implementiert werden, da freie Parameterlisten (ellipses) in Templates nicht erlaubt waren. Dies ist zwar kein Fehler des Standards selbst, schränkt aber die möglichen Plattformen ein und kann bei der Implementierung zu Problemen führen.
- Die oql-Funktionen sind unvollständig.
  Sowohl die möglichen Rückgabewerte als auch die Parameter der oql-Funktion sind auf wenige Typen eingeschränkt. Sie umfassen weder den vollen Sprachumfang des generi-

schen Teils noch die in der Sprachbindung verfügbaren Typen. Als Rückgabewerte sieht der Standard nur die sechs Datentypen int, char, double, char*, Ref<T> und Collection<T> vor, während allein im generischen Teil 23 verschiedene Datentypen definiert werden. Dieselbe Typliste wird auch bei der Parameterübergabe verwendet, ist dort aber weniger problematisch, da die Daten bei der Übergabe in den passenden allgemeineren Typ umgewandelt werden können. Man muß bei der Übergabe aber die automatischen Typkonvertierungen von C++ berücksichtigen (siehe Anhang H.7).

- Fehlendes Metaschema
  Der Standard definiert kein Metaschema für die Sprachbindung und C++ bietet ebenfalls keine Typinformationen zur Laufzeit. Die *Run-Time Type Identification (RTTI)* ([Str00]) bietet sich als zukünftige Lösung an.
- Fehlende Angaben zur Implementierung von Relationships
  Bei der Abbildung von Relationships erweitert der Standard die Syntax von C++. Es bleibt offen, wie Relationships letztendlich in reinem C++ dargestellt werden sollen. Deshalb habe ich meine eigenen Klassen entwickelt.
- Fehlende Nullwerte
  Im Gegensatz zum generischen Teil des Standards sind in der C++-Sprachbindung keine Nullwerte mehr vorgesehen. Nach Auskunft von Guy Ferran[50] war ursprünglich auch keine Änderung in dieser Richtung seitens der ODMG geplant. Eine Implementierung unseres Sichtsystems muß folglich ohne Nullwerte auskommen, was die Funktionalität stark einschränkt. In Version 2.0 des Standards ist inzwischen eingeschränkte Nullwertfunktionalität eingeführt worden (siehe Abschnitt 6.2.2).
- Fehlende Zugriffsfunktionen
  Die Funktionen get_value und set_value sind nicht in die C++-Sprachbindung übernommen worden. Die Verwendung von Zugriffsmethoden ist aber in C++ ein übliches Vorgehen, so daß wir dennoch auf sie zurückgreifen.
- Das Typsystem paßt nicht zu ODL
  Der ODL-Datentyp sequence erscheint nicht in der Sprachbindung, dafür gibt es für die C++-Klassen Date, Time, Timestamp und Interval keine Entsprechung in ODL. Die Klasse String ist offensichtlich definiert worden, um ODL-Strings darzustellen; sie wird aber nicht verwendet, da ODL-Strings direkt auf char* abgebildet werden. Weiterhin stimmen die in den vordefinierten Klassen angegebenen Methoden nicht mit den entsprechenden Methodennamen des Objektmodells überein.
- Mangelnde CORBA-Konformität
  Während die ODL-Syntax deutlich an IDL angelehnt ist, gibt es sowohl im Modell als auch in der Sprachbindung Unterschiede. Der IDL/ODL-Datentyp sequence etwa erscheint weder im Modell noch in der Sprachbindung. Die Abbildung der Basisdatentypen unterscheidet sich ebenfalls, da CORBA statt der konkreten C++-Datentypen die Verwendung generischer Typbezeichner wie etwa CORBA::ULong vorsieht. Der Zusammenhang zwischen den aus CORBA bekannten Arrays fester Größe (type[n]) und den neuen ODMG-Arrays variabler Größe (array<type>) wird nicht geklärt.

Ein weiterer Kritikpunkt betrifft die Rolle der Spezifikationssprache C++-OML. Gemäß der Zielsetzung von Seite 77 soll die Schemaspezifikation nahtlos in die C++-Syntax integriert werden. Aus diesem Grunde werden Interface-Spezifikationen direkt als C++-Klassendeklaratio-

50. Diskussion nach dem Vortrag „The Object Database Standard: ODMG–93“ von Guy Ferran (VP Engineering bei O_2 Technology) am 21. Oktober 1994 am Wissenschaftlichen Zentrum der IBM Deutschland in Heidelberg.

nen notiert. Um die Funktionalität von ODL vollständig darstellen zu können, wird die C++-Syntax zusätzlich um Schlüsselworte wie schema oder inverse erweitert. Die so erhaltene Sprache wird C++-ODL genannt. Sie hat allerdings den Nachteil, daß sie nicht mehr von einem normalen C++-Compiler verarbeitet werden kann. Man benötigt entweder einen erweiterten Compiler, oder man muß in einer Präkompiliationsphase alle nicht C++-konformen Erweiterungen durch C++-Konstrukte ersetzen. Diese Problematik stellt jedoch den Sinn von C++-ODL in Frage, da sie beiden Zielsetzungen des Standards — Integration und Austauschbarkeit — widerspricht. Die Integration nach C++ ist nicht gelungen, da sprachfremde Elemente verwendet werden. Wenn ein Datenbankschema gemäß der Sprachbindung direkt in C++-ODL spezifiziert wird, geht außerdem die Sprachunabhängigkeit und damit die Austauschbarkeit zwischen verschiedenen Plattformen verloren. Damit kann C++-ODL eigentlich nicht als Schemadefinitionssprache im Sinne des Standards fungieren. Das einzig sinnvolle Vorgehen in diesem Zusammenhang ist, das Schema in ODL zu spezifizieren und dann durch einen ODL-Präprozessor nach C++-ODL konvertieren zu lassen. Hierbei verliert C++-ODL allerdings vollständig seinen Sinn. Wird der generierte Code vom Benutzer nachträglich bearbeitet, so entspricht er nicht mehr der ursprünglichen ODL-Spezifikation — die Austauschbarkeit ist nicht mehr gegeben. Wird er nicht mehr bearbeitet, so muß er unverändert an einen weiteren Präprozessor übergeben werden, der dann eine reine C++-Ausgabe erzeugt. In diesem Fall wäre es aber wesentlich einfacher, die beiden Phasen zusammenzufassen und den C++-Code direkt aus der ODL-Spezifikation generieren zu lassen. Dies gilt besonders, da ein nicht-trivialer ODL-C++-Parser in der Lage sein muß, die komplexe C++-Syntax inclusive der Veränderungen durch cpp-Makros zu bearbeiten. Im allgemeinen kann man deshalb davon ausgehen, daß C++-ODL vollständig übersprungen wird, und aus einer ODL-Spezifikation direkt C++-Definitionen generiert werden. Dieser Ansatz der direkten C++-Generierung wird sowohl in der Ontos-ODB als auch in IRO-DB verfolgt. In ODMG 1.2 sind die neuen Schlüsselwörter wieder entfernt worden und C++-ODL ist als Standard-C++ unter Verwendung ODMG-spezifischer Datentypen definiert (siehe Abschnitt 6.2.1).

Sowohl die allgemeinen Probleme als auch die konkreten Probleme mit der Sprachbindung haben Einfluß auf den Entwurf und die Umsetzung des Sichtsystems. Manche Ideen waren hierdurch nicht umsetzbar.

Anhang H: Implementierungsdetails

Dieser Abschnitt beschreibt weitere Implementierungsdetails als Ergänzung zu Kapitel 4 der Arbeit. Abschnitt H.1 definiert die formale Abbildung von ODL nach C++. Dann folgen Algorithmen und Klassendefinitionen für die Objektverwaltung, die die Implementierung von Referenzen und Relationships beschreiben. In Abschnitt H.4 wird das Data Dictionary für die Darstellung von Metainformation definiert, in Abschnitt H.5 das generische Datenformat für die Darstellung von Zwischenergebnissen und in Abschnitt H.6 die einzelnen Hilfsfunktionen und Wrapper zum Zusammenspiel von Sprachbindung und Anfragebearbeitung.

H.1 Formale Abbildung von ODL nach C++

In Abschnitt 4.2 ist bereits eine Einführung in die C++-Sprachbindung gegeben worden. Die folgenden Funktionen und Tabellen definieren die vollständige verwendete Abbildung der ODL-Datentypen nach C++. Diese Abbildung ist nur eine von mehreren möglichen Interpretationen, da sie im Standard nicht konkret definiert ist sondern anhand einzelner inkonsistenter Beispiele beschrieben wird.

Tabelle 8 beschreibt die Abbildung für die literalen Datentypen. Fast alle atomaren Literaltypen liegen unter gleichem Namen auch in C++ vor. Zusätzlich wird wie üblich Boolean auf int abgebildet, Octet auf unsigned char und Any auf void*. Für Any könnte man auch dem CORBA-Standard folgen, der dem ungetypten Zeiger noch einen TypeCode hinzufügt.

Zur Abbildung der Collection-Typen werden eine Reihe von Template-Klassen definiert. Ausgenommen hiervon ist die Zeichensequenz, die einfach auf C++-Zeichenketten (char*) abgebildet wird. Aufzählungen und Strukturen können unverändert nach C++ übernommen werden. Für die Variante (**union**) ist keine Abbildung angegeben, und sie muß auch nicht von Implementierungen unterstützt werden ([Cat94, S. 90]). Da ODL-Varianten im Gegensatz zu C++ einen Diskriminator enthalten, folgt man bei der Abbildung am besten dem CORBA-Standard, der Diskriminator und Variante in einer Struktur zusammenfaßt.[51] Arrays fester Länge, die man durch Anhängen eckiger Klammern an einen Typ erhält, können direkt nach C++ übernommen werden. Arrays variabler Länge werden wie die anderen Collections durch Template-Klassen definiert. Für den im Standard definierten Sequence-Typ gibt es keine Entsprechung in der C++-Sprachbindung. Er ist aus IDL übernommen worden, wird in ODMG aber durch die spezialisierteren Formen ersetzt.

Atomare Objekttypen werden als Klassendefinitionen nach C++ übernommen (Tabelle 9). Attribute und Relationships werden zu Datenfeldern des Objekts. Die möglichen Datentypen der Attribute sind bereits beschrieben worden. Zur Darstellung von Relationships wird gemäß ODMG ein spezieller Referenzdatentyp Ref<T> definiert.[52] Unidirektionale Relationships zählen zu den Attributen und werden einfach durch ein Ref<>-Element repräsentiert. Für bidirektionale Relationships werden spezielle Relationship-Klassen implementiert, die sich neben der Navigation auch um die Zusicherung referenzieller Integrität kümmern (siehe Kapitel 4.4). Die im generischen Teil definierten Lese- und Schreiboperationen auf Attributen (get_value und set_value) sind in der offiziellen C++-Sprachbindung aufgegeben worden ([Cat94, S. 90]). Da dies für Sichten jedoch notwendig ist, werden Attribute auf private Datenfelder abbildet, auf die mit öffentli-

51. CORBA 2.0 verwendet eine etwas komplexere Typabbildung für Struct und Union, um mit Hilfe von Konstruktoren eine korrekte Initialisierung zu ermöglichen.

52. Siehe Kapitel 4.3.

chen Lese- und Schreibmethoden zugegriffen wird. Dieser Ansatz ist unter anderem auch von der ODMG-Library für Ontos verwendet worden („Ontos-ODB“ [GOP96]). Bei der Abbildung von Methoden muß zusätzlich auf die Richtung der Datenübergabe bei Parametern und Ergebniswerten geachtet werden.

Die Abbildung von Konstanten und benannten Typdefinitionen ist nicht definiert, die Konzepte sind aber identisch mit den C++-Elementen const und typedef. Auf Grund der geringen Verbreitung von Exception-Unterstützung bei den Compilern, empfiehlt der Standard die Verwendung einfacher Fehlerroutinen als Ersatzdarstellung.

Die beiden folgenden Definitionen und Tabellen beschreiben formal die Abbildung des ODMG-Typsystems nach C++:

Definition 9.1: CPP(T)

Die Abbildung CPP(T) liefert für einen ODMG-Datentyp T den zugehörigen C++-Datentyp der Sprachbindung. CPP(T) ist in den Tabellen 8 und 9 definiert.

Die C++-Datentypen Varray, Set, List, Bag und Ref werden im Standard definiert. Die Datentypen für die bidirektionalen Relationships sind selbstentwickelte Klassen. Sie sind im Implementierungskapitel näher erläutert.

	ODL-Datentyp T	**C++-Datentyp CPP(T)**
Atomare Literale	short	short
	long	long
	unsigned short	unsigned short
	unsigned long	unsigned long
	float	float
	double	double
	char	char
	boolean	int
	octet	unsigned char
	any	void*
	void	void
Strukturierte Literale	array<T>, array<T,n>	Varray<CPP(T)>
	sequence<T>, sequence<T,n>	*nicht verfügbar*
	set<T>	Set<CPP(T)>
	list<T>	List<CPP(T)>
	bag<T>	Bag<CPP(T)>
	string, string<n>	char*
	enum e { $name_1$, ... }	enum e { $name_1$, ... }
	struct s { T_1 $name_1$; ...; }	struct s { CPP(T_1) $name_1$; ...; }
	union u switch (T_s) { case v_1: T_1 $name_1$; ...; default: T_n $name_n$; }	struct u { CPP(T_s) _d; union { CPP(T_1) $name_1$; ...; CPP(T_n) $name_n$; } _u }
Arrays fester Länge	T[n]	CPP(T)[n]
Referenzen	OT	Ref<OT>

Tabelle 8: CPP(T) – Abbildung von ODL-Datentypen auf C++-Typen

	ODL-Datentyp T	**C++-Datentyp CPP(T)**
Objekttypen	Interface OT : ST_1, ..., ST_n { attribute AT_1 $attname_1$; ...; relationship RT_1 $relname_1$; ...; T methname (D_1 T_1 $name_1$,...); ...; }	class OT : public ST_1, ..., ST_n { public: CPP(AT_1) $attname_1$; ...; CPP(RT_1) $relname_1$; ...; virtual CPP_IO(RET,T) methname (CPP_IO(D_1,T_1) $name_1$,...); ...; }
Relationships (unidirektional)	OT	Ref<OT>
	set<OT>	Set<Ref<OT>>
	list<OT>	List<Ref<OT>>
	bag<OT>	Bag<Ref<OT>>
	array<OT>	Varray<Ref<OT>>
Relationships (bidirektional)	OT	SimpleRel<OT>
	set<OT>	SetRel<OT>
	list<OT>	ListRel<OT>
	bag<OT>	BagRel<OT>
	array<OT>	ArrayRel<OT>

Tabelle 9: CPP(T) – Abbildung von ODL-Objekttypen auf C++-Typen

Definition 9.2: CPP(IO,T)

Die Abbildung CPP(IO,T) liefert für einen ODMG-Datentyp T und eine Richtungsangabe IO {IN, OUT, INOUT, RET} den zugehörigen C++-Datentyp der Sprachbindung für Methodenparameter und Rückgabewerte. CPP(IO,T) ist wie folgt definiert:

$$\text{CPP(IN,T)} = \begin{cases} \text{const char*} & \text{für T = string} \\ \text{CPP(T)} & \text{für T} \quad \text{ODL\_atom} \quad \{ \text{OT, enum} \} \\ \text{const CPP(T)\&} & \text{für T} \quad \{ \text{array, set, list, bag, struct, union} \} \end{cases}$$

$$\text{CPP(OUT,T)} = \text{CPP(T)\&}$$

$$\text{CPP(INOUT,T)} = \text{CPP(T)\&}$$

$$\text{CPP(RET,T)} = \text{CPP(T)}$$

mit

ODL_atom = { short, long, unsigned short, unsigned long, float, double, char, boolean, octet, any }

H.2 Die Objektverwaltung

Die folgenden beiden Unterabschnitte beschreiben die Algorithmen und Implementierungsklassen für die Realisierung des Referenzmechanismus aus Kapitel 4.3.

H.2.1 Algorithmen

Für die nachfolgenden Algorithmen sind die folgenden Basisfunktionen zur Konvertierung und Zuordnung von Indexwerten, Schlüsselwerten und Objekten definiert:

xrid(r): Ref_Any→XRID

liefert die in einer Referenz abgelegte erweiterte Referenznummer.

ref(xrid): XRID→Ref_Any

erzeugt ein neues Ref_Any-Objekt mit der gegebenen XRID.

obj(xrid): XRID→Pobject

liefert das zur erweiterten Referenznummer gehörige Objekt bzw. NULL, wenn es nicht aktiviert ist. Hierzu führt jedes Verwaltungsobjekt ein internes Array oder eine Hashtabelle,

in der für jede XRID der zugehörige Objektzeiger eingetragen wird. Der Aufruf von obj(xrid) führt also nicht automatisch zu einer Aktivierung.

key(xrid): XRID→KeyType
: liefert die zur erweiterten Referenznummer gehörige Schlüsselinformation, sofern eine solche im zugehörigen Verwaltungsobjekt vorliegt. Der Ergebnistyp KeyType hängt vom jeweiligen Verwaltungsobjekt ab. Die Schlüsselinformation kann ebenfalls in der von obj(xrid) verwendeten Tabelle abgelegt werden.

xrid(vid,key): VID KeyType→XRID
: Inverse von key(xrid); sucht im durch vid indizierten Verwaltungsobjekt nach einem Eintrag mit dem Schlüsselwert key und liefert die zugehörige erweiterte Referenznummer bzw. die Nullreferenz, wenn kein solcher Schlüssel existiert. Um eine lineare Suche in der Tabelle key(xrid) zu vermeiden, verwendet das Verwaltungsobjekt hierfür eine zusätzliche Hashtabelle.

xrid(o): Pobject→XRID
: Inverse von obj(xrid); liefert die erweiterte Referenznummer eines Objekts. Anstatt eine weitere Tabelle anzulegen, wird in jedem Objekt zum Zeitpunkt seiner Erzeugung eine Referenz auf sich selbst eingetragen, die durch die Funktion ausgelesen wird (Pobject::_selfRef, siehe Anhang H.7).

Algorithmus zur Referenzerzeugung

Der folgende Algorithmus faßt alle Fälle der Referenzerzeugung zusammen: a) Kopie einer Referenz, b) Kopie der Selbstreferenz eines Objekts, c) Erzeugung aus einem Schlüsselwert. Der Sonderfall der schlüsselfreien Referenzerzeugung ist hier nicht explizit berücksichtigt; wenn es sich im letzten Teil des Algorithmus um die Hauptspeicher-Objektverwaltung handelt, entfällt die Schlüsselbearbeitung und es wird direkt die neue Referenz zurückgegeben.

```
Ref_Any create_reference (arg) {          // erzeugt eine neue Referenz für das gegebene Argument
   if arg    Ref_Any then                 // erzeuge eine direkte Kopie
      return ref(xrid(arg));
   if arg    Pobject then                 // hole die xrid aus der Selbstreferenz des Objekts
      return ref(xrid(o._selfRef));
   if arg is a key value for RefHdlr vid then   // suche nach dem Schlüssel im Verwaltungsobjekt vid
      if id=xrid(vid,arg) is defined then
         return ref(id);
      else
         create a new reference id (vid,++_currentId);
         insert the triple (vid,arg,id) into the table "xrid(vid,key)";
         insert the pair (id,key) into the table "key(xrid)";
         return ref(id); };
```

Algorithmus zur Dereferenzierung

Der folgende Algorithmus liefert das Objekt zu einer Referenz.

```
Pobject* get_object (xrid) {                          // liefert das zu einer XRID gehörige Objekt
   if obj(xrid) is not null then                      // das Objekt ist bereits aktiviert
      return obj(xrid);
   else
      o=activate_object(key(xrid));                   // aktiviere das Objekt
      o._selfRef=ref(xrid);                           // setze die Selbstreferenz
      insert the pair (xrid,o) into the table "obj(xrid)";   // Verwaltung aktualisieren
      return o; };
```

Aktivierung eines Importobjekts

Der folgende Algorithmus beschreibt die Aktionen zur Aktivierung eines Importobjekts:

```
Pobject* ImpRefHdlr::activate_object (goid) {          // aktiviert ein Importobjekt
    analyse the goid to determine the name of the object's most specific class from the DD;
    compute the name of the corresponding activation wrapper;
    connect to the external host;
    send a command to retrieve the object's properties;
    call the wrapper and pass the retrieved attribute values to it;
    return the resulting object; };
```

Aktivierung eines Konvertierungsobjekts bei Orig-Generalisierung

Die folgenden Aktionen sind für die Aktivierung eines Konvertierungsobjekts notwendig, wenn statt der sofortigen Instanziierung ein Aktivierungsmechanismus mit einem generischen orig-Typ verwendet wird:

```
Pobject* DerivRefHdlr::activate_object (key) {         // aktiviert ein Konvertierungsobjekt
    determine the object's class from the key value;
    compute the name of the corresponding activation wrapper;
    call the wrapper and pass the orig value from the key to it;
    return the resulting object; };
```

Deaktivierung

Bei der Deaktivierung eines Importobjekts müssen Änderungen zurückgeschrieben und die Objekttabelle angepaßt werden:

```
void deactivate_object (Pobject* obj) {                // deaktiviert ein Importobjekt
    if obj is modified then writeback changes to the external host;
    insert the pair (xrid,null) into the table "obj(xrid)";
    destruct obj; };
```

H.2.2 Klassen

Ref_Any und Ref<T> stellen die Referenzen dar, RefHdlr und seine Subklassen MemRefHdlr und ImpRefHdlr sind die zugehörigen Verwaltungsobjekte. Drei Objektmanager OM, IOM und DOM stellen eine vereinfachte Schnittstelle zur Objektverwaltung zur Verfügung.

Ref_Any

Ref_Any ist die in ODMG–93 definierte Implementierung für ungetypte Objektreferenzen. Neben den in ODMG–93 geforderten Funktionen sind noch weitere Abfrage- und Managementfunktionen implementiert. Die Implementierung basiert auf einer Referenznummer (_refId), die von einem zugehörigen Managerobjekt (RefHdlr) vergeben und verwaltet wird. Mit is_null und is_active kann getestet werden, ob die Referenz auf ein Objekt zeigt, und ob sich dieses Objekt bereits im Speicher befindet. An das Objekt selbst kommt man durch Aufruf von ptr bzw. durch einen Cast auf (Pobject*). metaClass liefert die Typinformation über das referenzierte Objekt; soweit möglich geschieht dies ohne Aktivierung des Objekts. check_type überprüft auf Zuweisungskompatibilität mit einer anderen Klasse. deactivate entfernt das Objekt aus dem Speicher und delete_objekt löscht das Objekt aus der Datenbank. Eine Reihe von Vergleichsoperatoren sind als freie Funktionen implementiert. Eine globale Ref_Any-Instanz nil steht als Standard-Nullreferenz zur Verfügung. Die meisten Funktionen delegieren die Aufgaben an das zugehörige Verwaltungsobjekt.

```
typedef unsigned long RefIdType;

class Ref_Any {
    protected:   unsigned int    _handler: 3;
                 unsigned int    _refId    : 29;
```

```
                unsigned short  handler () const;
                RefHdlr*        get_handler () const;
                RefIdType       refId () const;
                                Ref_Any (unsigned short handler, RefIdType refId);
                void            check_type (InterfaceDef* super) const;
                void            deactivate ();
    public:                     Ref_Any ();
                                Ref_Any (const Ref_Any& other);
                                Ref_Any (Pobject* obj);
                                Ref_Any (Pobject& obj);
                                ~Ref_Any ();
                Ref_Any&        operator= (const Ref_Any& other);
                Ref_Any&        operator= (Pobject* obj);
                Ref_Any&        operator= (Pobject& obj);
                void            clear ();
                int             is_null () const;
                int             is_active () const;
                void            delete_object ();
                                operator Pobject*() const;
                Pobject*        ptr () const;
                InterfaceDef*   metaClass () const;
    static      const Ref_Any   nil; };
```

Ref<T>

Die Ref-Klasse ist wie im Standard beschrieben ein Template, das einen typsicheren Zugriff auf Ref_Any-Instanzen zur Verfügung stellt. Die Funktionen sind vollständig als inline implementiert, um den Compiler-Aufwand zu reduzieren. Im Projekt haben wir die Erfahrung gemacht, daß die kombinatorische Vielfalt bei der Instanziierung und die gegenseitige Abhängigkeit zwischen Templates zu einem inflationären Anwachsen der benötigten Template-Instanziierungen führt. Durch die hierfür notwendige Codegenerierung, Abhängigkeitsprüfung und Übersetzung ist die Kompilationszeit teilweise auf das Zehnfache angestiegen. Dies kann durch konsequente Nutzung von inline-Code vermieden werden.

```
template <class T> class Ref : public Ref_Any {
    public:     inline              Ref ();
                inline              Ref (T* r);
                inline              Ref (T& r);
                inline              Ref (const Ref<T>& r);
                inline              Ref (const Ref_Any& r);
                inline              ~Ref ();
                inline Ref<T>&      operator= (T* r);
                inline Ref<T>&      operator= (T& r);
                inline Ref<T>&      operator= (const Ref<T>& r);
                inline Ref<T>&      operator= (const Ref_Any& r);
                inline              operator Ref_Any() const;
                inline              operator T*() const;
                inline T*           ptr () const;
                inline T*           operator–> () const;
                inline T&           operator* () const; };
```

RefHdlr

RefHdlr ist die abstrakte Basisklasse für alle Objektverwaltungsklassen. Sie verwaltet die Vergabe von Referenznummern und deren Zuordnung zu Objekten. Der erste Block enthält Funktionen zur Verwaltung der globalen Liste von Verwaltungsobjekten. _handler enthält die Liste, _defaultHdlr zeigt auf die standardmäßig zu verwendende Hauptspeicher-Verwaltung, und die angegebenen Funktionen dienen zum Hinzufügen, Löschen und Abfragen von Objekten aus der Liste. Um das duplizieren einer Instanz zu verhindern sind der Kopierkonstruktor und der Zuweisungsoperator privat und führen bei Aufruf zu einem Fehler. Das Datenfeld _index enthält die eigenen Verwaltungsobjektnummer VID, _currentId enthält die höchste eigene bisher vergebene Referenznummer. Die Funktionen getNewRefId, ptr, metaClass und isActive werden an die Subklassen vererbt. Sie stellen allgemeine Basisfunktionalität und Alternativnamen für die abstrak-

ten Funktionen zur Verfügung. Die abstrakten Funktionen createRef, lookup, instantiate, deleteObject und deactivate dienen der internen Zuordnungsverwaltung und müssen von den Subklassen neu implementiert werden. Sie realisieren die konkreten Zugriffe auf die Verwaltungstabellen des Objekts. Die einzige öffentliche Funktion ist das statische ref, das für ein bereits existierendes Objekt eine Referenz erzeugt. Hat dieses Objekt noch keine Selbstreferenz, die kopiert werden könnte, wird die Standardverwaltung verwendet. Alle anderen Funktionen sind geschützt und können nur indirekt über Ref_Any oder andere Objektverwaltungsklassen aufgerufen werden.

```
class RefHdlr {
   private:                          RefHdlr (const RefHdlr&);
                RefHdlr&             operator= (const RefHdlr&);
   protected:   static RefHdlr*      _handler [MAXREFHANDLER+1];
                static RefHdlr*      _defaultHdlr;
                static unsigned short registerHandler (RefHdlr* hdlr);
                static void          unregisterHandler (unsigned short idx);
                static RefHdlr*      getHandler (unsigned short idx);
                static void          createDefaultHdlr ();
                static RefHdlr*      getDefaultHdlr ();
                                     RefHdlr ();
                virtual              ~RefHdlr ();
                unsigned short       _index;                          // own VID
                RefIdType            _currentId;                      // highest RID
                RefIdType            getNewRefId ();                  // next higher RID
                Pobject*             ptr (RefIdType refId);           // pointer to ref'd object
                virtual InterfaceDef* metaClass (RefIdType refId);    // metaclass of ref'd object
                int                  isActive (RefIdType refId) const; // is ref'd object in memory?
                virtual Ref_Any      createRef (Pobject* obj) = 0;    // new ref for object
                virtual Pobject*     lookup (RefIdType refId) const = 0; // find object for ref
                virtual Pobject*     instantiate (RefIdType refId);   // activate the object
                virtual void         deleteObject (RefIdType refId);  // remove object from DB
                virtual void         deactivate (RefIdType refId);    // deactivate object
   public:      static Ref_Any       ref (Pobject* obj); };
```

MemRefHdlr

MemRefHdlr ist die Hauptspeicher-Objektverwaltung, die für alle Objekte verwendet wird, die kein spezifischeres Verwaltungsobjekt verwenden. Es verwendet nur eine Hashtabelle, die von Referenznummern auf Objektadressen verweist. Ein Array würde zu viel Platz verbrauchen und immer nur gering gefüllt sein. Eine Schlüsselverwaltung ist nicht notwendig, da der MemRefHdlr keine Objektaktivierung unterstützt.

```
class MemRefHdlr : public RefHdlr {
   private:                          MemRefHdlr (const MemRefHdlr&);
                MemRefHdlr&          operator= (const MemRefHdlr&);
   protected:   om_dict<RefIdType,Pobject*>* _dict;                  // hash table rid→Pobject
                                     MemRefHdlr ();
                virtual              ~MemRefHdlr ();
                virtual Ref_Any      createRef (Pobject* obj);
                virtual Pobject*     lookup (RefIdType refId) const;
                virtual void         deleteObject (RefIdType refId);
                virtual void         deactivate (RefIdType refId); };
```

ImpRefHdlr

ImpRefHdlr ist das Verwaltungsobjekt für Importobjekte. Als Schlüsselwerte werden globale OIDs verwendet. Entsprechend gibt es zwei Hashtabellen goid→rid und rid→(goid,Pobject), wobei die letzte Zielstruktur als eigene Klasse ObjectInfo implementiert ist. Sie enthält eine GOID, die zugehörige Objektadresse und einen Zeiger auf die Klasseninformation; mit invalidate wird die Objektadresse nach einer Deaktivierung gelöscht.. In der Tabelle instCount zählt der ImpRefHdlr für jede Datenbank die derzeit aktivierten Objekte. Hierdurch kann man den Rück-

schreibvorgang beim Transaktionsende oder bei Methodenaufrufen beschleunigen. Zusätzlich zur Implementierung der geforderten abstrakten Funktionen werden weitere Hilfs- und Zugriffsmethoden definiert. incCount und decCount aktualisieren den Instanzzähler der angegebenen Datenbank. markInactive löscht nach einer Deaktivierung die alte Objektadresse aus der Suchtabelle, markInvalid wird nach einer Objektlöschung aufgerufen und löscht zusätzlich die eingetragene GOID. insertDictEntry erzeugt einen neuen Tabelleneintrag für die GOID und liefert die zugewiesene RID zurück. ldaId und goidForRefId suchen die entsprechenden Einträge in den Tabellen. metaClass ist eine Reimplementierung von RefHdlr::metaClass, die eine Aktivierung des Objekts verhindert, da die Klasseninformation aus der GOID bestimmt werden kann. writebackObject schreibt geänderte Objektattribute in die externe Datenbank zurück. Öffentlich aufrufbar sind neben den Schlüsselzuordnungsfunktionen refForGoid und goidForRef drei weitere Funktionen: getCount liefert den Instanzzähler zu einer externen Datenbank; writebackLocal schreibt sämtliche Objektänderungen einer externen Datenbank zurück und deaktiviert gegebenenfalls die Objekte; cleanup deaktiviert sämtliche Importobjekte, wobei der Parameter invalidate das Zurückschreiben von Änderungen verhindert.

```
class ImpRefHdlr : public RefHdlr {
    private:                                ImpRefHdlr (const ImpRefHdlr&);
                    ImpRefHdlr&             operator= (const ImpRefHdlr&);
    protected:      om_dict<char*,RefIdType>  _refInfo;                    // goid→rid
                    om_dict<RefIdType,ObjectInfo> _importObjInfo;          // rid→(goid,Pobject)
                    om_dict<const char*,int>  instCount;                   // ldaid→count
                                            ImpRefHdlr ();
                    virtual                 ~ImpRefHdlr ();
                    virtual Ref_Any         createRef (Pobject* obj);
                    virtual Pobject*        lookup (RefIdType refId) const;
                    virtual Pobject*        instantiate (RefIdType refId);
                    virtual void            deleteObject (RefIdType refId);
                    virtual void            deactivate (RefIdType refId);
                    void                    incCount (const char* ldaid);  // increment DB obj counter
                    void                    decCount (const char* ldaid);  // decrement DB obj counter
                    void                    markInactive (RefIdType refId);// mark obj as deactivated
                    void                    markInvalid (RefIdType refId); // mark object as invalid
                    RefIdType               insertDictEntry (const char* goid); // create new entry
                    virtual const char*     ldaId (RefIdType refId);       // return the ldaid for the rId
                    virtual const char*     goidForRefId (RefIdType refId);// return goid (key) for rid
                    virtual InterfaceDef*   metaClass (RefIdType refId);   // return class info w/o activ.
                    virtual void            writebackObject (RefIdType refId); // write object to DB
    public:         virtual Ref_Any         refForGoid (const char* goid, InterfaceDef* impcls); // goid→ref
                    virtual const char*     goidForRef (const Ref_Any& ref);       // ref→goid
                    int                     getCount (const char* ldaid) const;    // return DB obj counter
                    virtual int             writebackLocal (const char* ldaid, bool remove); // put updates
                    virtual int             cleanup (bool invalidate); };          // deactivate all objects

class ObjectInfo {
    protected:      char*                   _goid;                         // global oid of the external object
                    Pobject*                _ptr;                          // pointer to the import object
                    InterfaceDef*           _metaClass;                    // pointer to the import class
    public:                                 ObjectInfo (const char* goid, Pobject* obj = 0);
                                            ObjectInfo (const ObjectInfo& other);
                                            ~ObjectInfo ();
                    void                    invalidate (); };
```

Object Manager OM

Der *Objektmanager* dient als Schnittstelle zwischen der Objektverwaltung und allen anderen Modulen im System. Er stellt alle geforderten Objekt-Funktionen zur Verfügung und delegiert sie gegebenenfalls an die Objektmanager für Import- und Konvertierungsobjekte, die bei der Initialisierung in den Datenfeldern _iom und _dom abgelegt werden. Für die Klasse OM wird im System genau ein Objekt angelegt, indem der Konstruktor in der statischen init-Methode aufgerufen wird und das erzeugte OM-Objekt in der globalen Variable theOM zur Verfügung gestellt wird. Mit getCurrentExtent[Of] und getFullExtent[Of] kann für ein Klassenobjekt oder einen Extent-Na-

men die Menge der aktuell instanziierten Objekte bzw. sämtliche zum Extent gehörigen Objekte bestimmt werden. Hierzu hält jede Klasse ihren aktiven Extent in einer Instanzmenge, die automatisch durch die Konstruktoren und Destruktoren aktualisiert wird; zur Berechnung des vollen Extents wird die Extentklausel ausgewertet. Mit getProperty und setProperty kann generisch ein Attribut oder eine Relationship eines beliebigen Objekts ausgelesen bzw. gesetzt werden. Mit executeMethod kann man Methoden auf Objekten ausführen. Zur Parameterübergabe dient die Hilfsklasse ALHolder (*Argument List Holder*), die die Parameter sowohl in Listen- als auch in Strukturform enthält. Mit getObject wird ein existierendes Objekt aktiviert und zurückgeliefert, mit createObjekt wird ein neues Objekt in der Datenbank erzeugt und mit deleteObject wird ein Objekt aus der Datenbank entfernt. Die letzte Operation entspricht dem destroy-Kommando von ODMG–93. Die writeback-Funktion dient dazu, geänderte Objekte in die externe Datenbank zurückzuschreiben und sie gegebenenfalls im Sichtsystem zu deaktivieren. Diese Funktion muß aufgerufen werden, wenn eine Transaktion endet, oder wenn direkt auf die externe Datenbank zugegriffen wird, wie etwa bei Methodenaufrufen oder Anfrageauswertungen in der externen Datenbank. Die sogenannten *Callback-Funktionen* dienen der direkten Interaktion mit anderen Modulen. Die Funktionen openCallback, closeCallback, beginCallback, writebackCallback und invalidateCallback werden von der Transaktionsverwaltung aufgerufen, wenn eine Transaktion anfängt oder endet. Sie initialisieren oder bereinigen den Objekt-Cache. Die Funktionen globalize und localize werden von der Netzwerkkomponente verwendet, um die bei der Datenübertragung verwendeten GOIDs in Referenzen umzuwandeln. Diese Funktionen sind als Callbacks implementiert, das heißt, sie werden erst zur Laufzeit bei den jeweiligen Modulen registriert. Zur Kompilationszeit ist nur ihre Signatur bekannt. Hierdurch können zyklische Abhängigkeiten zwischen den Softwaremodulen verhindert werden, was die Modulentwicklung und -übersetzung vereinfacht. Die internen Funktionen execActivationConstructor und getAllProps werden zur Objektaktivierung und zur Datenübertragung bei writeback benötigt.

```
class OM {
   private:          OM (const OM&);
                     OM& operator= (const OM&);
   protected:        IOM*        _iom;                                  // the corresponding IOM
                     DOM*        _dom;                                  // the corresponding DOM
                                 OM ();
                                 ~OM ();
                     static int  openCallback (void* arg);              // after db open
                     static int  closeCallback (void* arg);             // before db close
                     static int  beginCallback (void* arg);             // after transaction begin
                     static int  writebackCallback (void* arg);         // write objects to db
                     static int  invalidateCallback (void* arg);        // discard changes
                     Ref_Any     globalize (const char* goid);          // get reference for goid
                     const char* localize (const Ref_Any& ref);         // get goid for reference
                     Pobject*    execActivationConstructor (InterfaceDef* clsname, Database* db, Activator& act);
                                                                        // call activation constr., init props
                     GFStruct*   getAllProps (Pobject* obj);            // read all object properties
   public:           static void init ();                               // initialize the Object Manager
                     void        finish ();                             // shut down the Object Manager
                     Set<Ref_Any>* getCurrentExtentOf (InterfaceDef* cls);   // current class instances
                     Set<Ref_Any>* getCurrentExtent (const char* extname);   // current extent instances
                     Set<Ref_Any>* getFullExtentOf (InterfaceDef* cls);      // all class instances
                     Set<Ref_Any>* getFullExtent (const char* extname);      // all extent instances
                     GFAny       getProperty (Ref_Any ref, PropertyDef* prop);          // read a property
                     void        setProperty (Ref_Any ref, PropertyDef* prop, GFAny val); // write prop.
                     GFAny       executeMethod (Ref_Any ref, OperationDef* op, ALHolder& args);
                     Ref_Any     getObject (InterfaceDef* cls, GFAny origInfo);         // activate objext
                     Ref_Any     createObject (InterfaceDef* cls, ALHolder& props);     // create object
                     void        deleteObject (Ref_Any ref);                            // delete object
                     void        writeback (const char* ldaId, bool remove); };         // write objects to db

extern OM* theOM;      // the global Object Manager instance
```

```
class ALHolder {
   protected:  GFStruct*      _struct;                          // the struct variant
               List<GFAny>*   _list;                            // the list variant
               int            _initial;                         // the initial value: 0=_struct, 1=_list
   public:                    ALHolder (GFStruct* st);
                              ALHolder (List<GFAny>* lst);
                              ALHolder (const ALHolder& other);
               ALHolder&      operator= (const ALHolder& other);
                              ~ALHolder ();
               bool           hasStruct () const;               // true, when the GFStruct value is available
               bool           hasList () const                  // true, when the List value is available
               GFStruct*      getStruct ();                     // return the GFStruct value, convert it if necessary
               List<GFAny>*   getList () const;                 // return the List value; conversion is not done
               void           createList (List<char*>* order); }; // explicit conversion from GFStruct to List according to the
                                                                // given attribute order
```

Import Object Manager IOM

Der *Import Object Manager* IOM wird vom OM verwendet, um auf Importobjekte zuzugreifen. Es darf nur ein IOM-Objekt vom OM erzeugt werden; die Konstruktoren sind deshalb von außen nicht zugreifbar. Das Datenfeld _refHdlr zeigt auf die Import-Objektverwaltung. executeMethod führt die notwendigen Operationen für eine Methodenausführung in der externen Datenbank aus: geänderte Objekte werden zurückgeschrieben, aus dem Methodennamen und den übergebenen Parametern wird eine OQL-Anfrage erzeugt und der externen Datenbank zur Auswertung übergeben, schließlich werden die Ergebnisse zurückgegeben. createObject erzeugt ein neues Importobjekt, indem aus den übergebenen Parametern ein OQL-Konstruktoraufruf erzeugt wird, mit dem die externe Datenbank das neue Objekt generieren kann. Die aus der zurückgelieferten GOID erzeugte Import-Objektreferenz wird an den Aufrufer zurückgegeben.

```
class IOM {
   private:                   IOM (const IOM&);
               IOM&           operator= (const IOM&);
   protected:  ImpRefHdlr*    _refHdlr;                                  // handler for import references
                              IOM ();
                              ~IOM ();
   public:     GFAny          executeMethod (Ref_Any ref, OperationDef* op, ALHolder& args);
                                                                         // execute method on local object
               Ref_Any        createObject (InterfaceDef* cls, ALHolder& props); };   // create local object
```

Derived Object Manager DOM

Der *Derived Object Manager* DOM übernimmt die Interaktion mit Konvertierungsobjekten. Auch vom DOM wird nur ein Objekt im OM erzeugt; die Konstruktoren sind nicht von außen zugänglich. Da wir keine eigene Konvertierungs-Objektverwaltung haben, übernimmt der DOM mit der Funktion getObject die Suche nach Konvertierungsobjekten, wobei durch Aufruf von priv_hasKey überprüft wird, ob ein bereits existierendes Objekt dem gesuchten Konvertierungsobjekt entspricht. Wird ein Objekt gefunden, so wird es zurückgegeben, anderenfalls wird ein neues Konvertierungsobjekt angelegt. Die eigentliche Suche erfolgt in der internen Funktion lookupObject. Mit deactivateObject wird der DOM informiert, daß ein Objekt deaktiviert worden ist. Mit createObject wird ein neues Konvertierungsobjekt erzeugt, indem die zugehörige Objekterzeugungsfunktion create_object aus der Sprachbindung aufgerufen wird.

```
class DOM {
   private:                   DOM (const DOM&);
               DOM&           operator= (const DOM&);
   protected:                 DOM ();
                              ~DOM ();
               Pobject*       lookupObject (InterfaceDef* cls, GFAny origInfo);   // check for existing obj.
               void           deactivateObject (Ref_Any ref);                     // mark obj. as deactiv.
   public:     Ref_Any        getObject (InterfaceDef* cls, GFAny origInfo);      // find or activate object
               Ref_Any        createObject (InterfaceDef* cls, ALHolder& props); }; // create new object
```

H.3 Referenzielle Integrität von Relationships

Diese Kapitel beschreibt die Implementierungsklassen für die einzelnen Relationship-Pfade und beschreibt den Algorithmus zur Zusicherung referenzieller Integrität mit Hilfe der beiden Basisfunktionen simpleRemove und simpleInsert.

H.3.1 Algorithmen

In Kapitel 4.4 habe ich beschrieben, daß durch die Definition der folgenden beiden Basisfunktionen eine weitgehende Unabhängigkeit zwischen den inversen Pfaden erreicht wird:

simpleRemove (target)
: Entfernt target aus der Menge der eigenen Ziele. Die inverse Relation von target zum eigenen Trägerobjekt wird nicht entfernt.

simpleInsert (target)
: Setzt target als neues Ziel dieses Pfades bzw. fügt es zur Menge oder Liste hinzu. Falls notwendig, werden zuvor bestehende Pfade entfernt; die inverse Relation von target zum eigenen Trägerobjekt wird hingegen nicht verändert.

Die Funktionen müssen für jede Wertigkeit und Klassenkategorie unterschiedlich implementiert werden, haben aber in allen Fällen dieselbe Signatur und können daher unabhängig von der Wertigkeit der Inversen bei der Aufdatierung aufgerufen werden. Die Funktionen werden durch die folgenden beiden Algorithmen beschrieben, wobei _target[s] das bisherige Zielobjekt bzw. die Menge oder Liste der bisherigen Zielobjekte darstellt und _myhost das eigene Trägerobjekt.

```
void simpleRemove (Ref_Any target) {
    if UnaryRelationship then _target = nil;                                  // einwertig: Referenz löschen
    if SetRelationship then     _targets.remove_element (target);             // mengenwertig: Referenz aus Menge löschen
    if ListRelationship then    _targets.remove_element_at ( findIndex(target) );  // listenwertig: erste Referenz aus Liste löschen
    _myhost.mark_modified (); }

void simpleInsert (Ref_Any target) {
    if UnaryRelationship then                                                 // einwertig:
        if _target != nil then                                                    // falls notwendig alte Pfade löschen:
            getInvRelFor(_target).simpleRemove (_myhost);                         // inversen Pfad löschen
            simpleRemove (_target);                                               // eigenen Pfad löschen
        _target = target;                                                         // neuen Pfad zuweisen
    if SetRelationship then                                                   // mengenwertig:
        _targets.insert_element (target);                                         // Pfad hinzufügen
    if ListRelationship then                                                  // listenwertig:
        _targets.insert_element_after (target, getCardinality()-1);               // Pfad hinzufügen
    _myhost.mark_modified (); }
```

Die Funktionen remove_element, remove_element_at, insert_element und insert_element_after sind hierbei einfache Operationen auf Referenzmengen bzw. -listen. getInvRelFor liefert das inverse Pfadobjekt, das im übergebenen Zielobjekt eingebettet ist. Der Übersichtlichkeit halber sind die notwendigen Konsistenzprüfungen nicht mit angegeben. Sie umfassen eine Typüberprüfung des zuzuweisenden Objekts, eine Existenzüberprüfung für zu löschende Objekte, sowie Sonderbehandlungen für die Zuweisung von Null-Referenzen und für reflexive Relationships. Die folgenden Beispiele zeigen, wie die Elementaroperationen zu Standardoperationen kombiniert werden können.

Traversierung:
: Beim Lesezugriff wird die im Pfadobjekt abgelegte Referenz, bzw. die Menge oder Liste von Referenzen an den Aufrufer zurückgegeben. Ein Zugriff auf die Inverse ist hierbei nicht notwendig. Die Elementaroperationen werden nicht verwendet.

Einfügen eines Objekts:

Beim Einfügen in eine mengenwertige Relationship wird zuerst die Referenz in die eigene Zielmenge eingetragen, dann teilt man der Inversen mit, daß das eigene Trägerobjekt in dessen Zielmenge aufgenommen werden soll. Der Aufruf von simpleInsert auf dem Zielobjekt führt dabei entsprechend der Wertigkeit der Inversen automatisch die notwendigen Operationen aus. Hierbei können auch weitere Objekte betroffen sein, die automatisch durch die Elementaroperationen angesprochen werden.

```
void SetRelImpl::insert (Ref_Any target) {
    simpleInsert (target);
    getInvRelFor(target).simpleInsert (_myhost); }
```

Entfernen eines Objekts:

Beim Löschen einer Verbindung müssen beide Pfade zurückgesetzt werden.

```
void SetRelImpl::remove (Ref_Any target) {
        getInvRelFor(target).simpleRemove (_myhost);
        simpleRemove (target); }
```

Zuweisung einer Objektmenge:

Bei der Zuweisung werden die Zielobjekte eines Pfades neu festgelegt. Zuerst müssen hierfür alle bestehenden Verbindungen zu den bisherigen Zielobjekten unterbrochen werden. Das Löschen der Verbindung geschieht dabei sowohl beim aktuellen Pfadobjekt als auch bei allen bisherigen Zielobjekten. Nach dieser Löschung werden die neuen Objekte eingefügt. Auch hierbei müssen wieder das Pfadobjekt und alle Zielobjekte verändert werden.

```
void ListRelImpl::set (List<Ref_Any> newtargets) {
    foreach t in _targets do remove (t);
    foreach t in newtargets do insert (t); }
```

Beispiel 9.1:

Die folgenden Code-Fragmente zeigen den vollständigen Ablauf der auf Seite 91 in Bild 13 gezeigten Adreßzuweisung mit dem automatischen Aufbrechen der alten Abhängigkeiten. Die Objektstrukturen kann man Bild 14 entnehmen. Gegeben seien die folgenden Objekte:

p1, p2		Person
a1, a2		Address
pa1=p1.address,	pa2=p2.address	SimpleRel <Address>
ar1=a1.resident,	ar2=a2.resident	SimpleRel <Person>
pai1=pa1._impl,	pai2=pa2._impl,	
ari1=ar1._impl,	ari2=ar2._impl	SimpleRelImpl

Gemäß Bild 13b ist zu Beginn der Person p1 die Adresse a1 und der Person p2 die Adresse a2 zugeordnet. Der Benutzer weist nun der Relationship address von Objekt p1 den neuen Wert a2 zu:

```
p1.address.set (a2)
```

Über das in p1.address abgelegte Zugriffsobjekt pa1 wird der Zugriff an das Implementierungsobjekt pai1 weitergeleitet:

```
pai1.set (a2)
```

Mit Hilfe von getInvRelFor (a2) und _myhost wird der inverse Pfad ari2, also die Implementierung von a2.resident, und das eigene Trägerobjekt p1 bestimmt. Hierdurch kann man simpleInsert auf beiden Pfaden aufrufen:

```
pai1.simpleInsert (a2)
ari2.simpleInsert (p1)
```

In beiden Fällen ist die Relationship schon mit einem Wert belegt, der zuerst gelöscht werden muß. a1 und p2 sind in den _target-Variablen abgelegt; an die zugehörigen Pfadimplementierungen kommt man wieder durch Ausführung von getInvRelFor.

```
// p1.address.set (a2)
    // pai1.set (a2)
        // pai1.simpleInsert (a2)
            ari1.simpleRemove (p1)
            pai1.simpleRemove (a1)
            pai1._target = a2
        // ari2.simpleInsert (p1)
            pai2.simpleRemove (a2)
            ari2.simpleRemove (p2)
            ari2._target = p1
```

Die ausgeführten simpleRemove-Operationen haben neben dem Löschen von _target keine weiteren Auswirkungen. Es werden also nacheinander alle alten Referenzen gelöscht und dann die neuen zugewiesen. ■

H.3.2 Klassen

Bild 29 zeigt die Hierarchie aller vordefinierten Relationship-Klassen. Der obere Teil enthält die typunabhängigen Zugriffsklassen für einwertige, mengenwertige und listenwertige Relationships (SimpleRelAny, SetRelAny, ListRelAny) sowie die davon abgeleiteten typspezifischen Templates (SimpleRel<T>, SetRel<T>, ListRel<T>). Jede dieser Klassen stellt die vom Standard geforderten Funktionen zur Verfügung und enthält einen Zeiger auf ein zugehöriges Implementierungsobjekt. Die typunabhängigen Zugriffsklassen arbeiten dabei auf ungetypten Referenzen (Ref_Any) und erhalten erst durch die C++-Templates eine Typhülle. Da die Templates außer Konvertierungen keinen eigenen Code erfordern und auch keine virtuellen Funktionen enthalten, können sie vom Compiler in Inline-Code umgesetzt werden, was zu einer extremen Performanzsteigerung zur Übersetzungszeit führt. In der unteren Hälfte von Bild 29 sind die zugehörigen Implementierungsklassen angegeben. Die DerivImpl-Klassen sind hierbei abstrakt und erfordern spezifische Subklassen für jede abgeleitete Relationship in der Sicht.

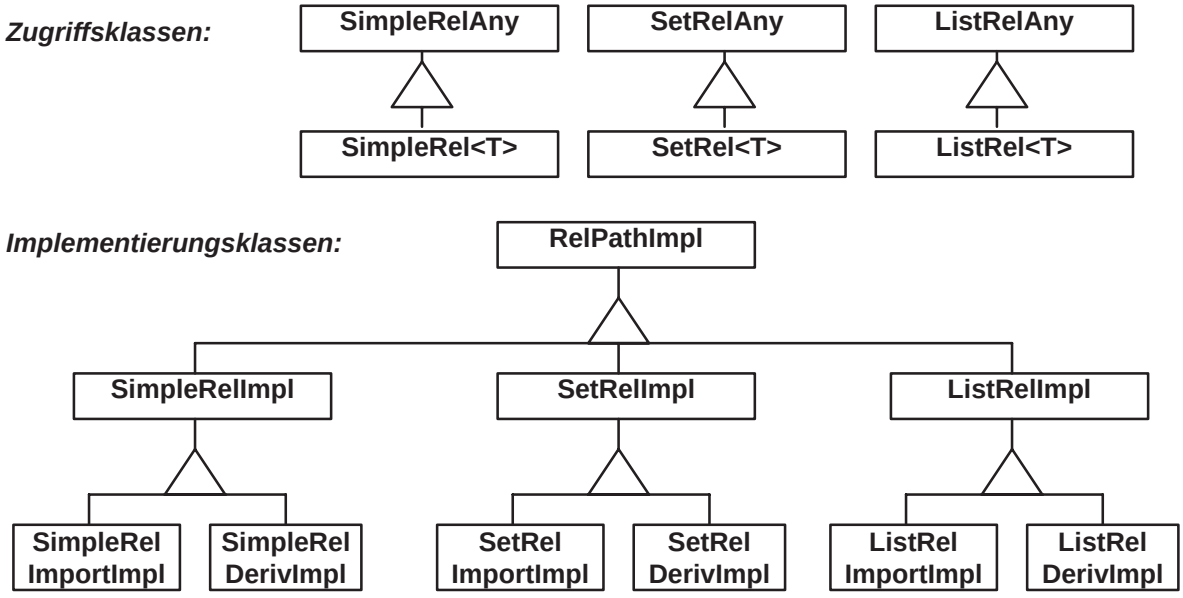


Bild 29: Klassenhierarchie zur Relationshipdarstellung

Zugriffsklassen

SimpleRelAny

SimpleRelAny ist die generische Zugriffsklasse für einwertige Relationships. Sie enthält einen Zeiger auf eine Implementierungsklasse vom Typ SimpleRelImpl und stellt die im Standard defi-

nierten Lese- und Schreibmethoden für eine einwertige, ungetypte Referenz zur Verfügung. Durch die Bidirektionalität der Relationships hat das Kopieren und Zuweisen einer Relationship keine saubere Semantik. Die entsprechenden Konstruktoren führen daher zu einer Fehlermeldung. Zuweisungen sind über (unidirektionale) Referenzen möglich, wobei sich die Semantik aus der verwendeten Einfügeoperation ergibt.

```
class SimpleRelAny {
   private:                 SimpleRelAny (const SimpleRelAny&);   // disabled
                SimpleRelAny& operator= (const SimpleRelAny&);   // disabled
   protected:   SimpleRelImpl* _impl;                            // the implementation object
                Ref_Any     privGet () const;                    // get target
                void        privSet (const Ref_Any& target);     // assign target
   public:      SimpleRelImpl* impl () const;                    // return implementation
                               SimpleRelAny (SimpleRelImpl* impl); // public constructor
                               ~SimpleRelAny ();                // public destructor
                               operator Ref_Any () const;       // cast operator
                int         is_null () const;                    // check for null reference
                void        clear ();                            // remove target
                void        delete_object (); };                 // delete target object
```

SetRelAny

SetRelAny stellt die generische Zugriffsklasse für ungeordnete mengenwertige Relationships dar. Die zugehörige Funktionalität wird von einem zugeordneten Implementierungsobjekt vom Typ SetRelImpl zur Verfügung gestellt. Genau wie bei einfachen Relationships werden Relationship-Zuweisungen und Copy-Konstruktoren nicht unterstützt.

```
class SetRelAny {
   private:                 SetRelAny (const SetRelAny&);                  // disabled
                SetRelAny&  operator= (const SetRelAny&);                  // disabled
   protected:   SetRelImpl* _impl;                                         // the implementation object
                Set<Ref_Any> privGet () const;                             // get target set
                void        privSet (const Set<Ref_Any>& targets);         // assign target set
                void        privInsert (const Ref_Any& target);            // add to target
                void        privRemove (const Ref_Any& target);            // remove from target
                int         privContains (const Ref_Any& target) const;    // check containment
                Iterator<Ref_Any> privIterator () const;                   // iterate over target set
   public:      SetRelImpl* impl () const;                                 // return implementation
                            SetRelAny (SetRelImpl* impl);
                            ~SetRelAny ();
                            operator Set<Ref_Any> () const;
                int         is_empty () const;
                unsigned long cardinality () const;
                void        clear ();
                void        remove_all (); };
```

ListRelAny

ListRelAny ist die Basiszugriffsklasse für geordnete mengenwertige Relationships. Das zugehörige Implementierungsobjekt ist vom Typ ListRelImpl.

```
class ListRelAny {
   private:                 ListRelAny (const ListRelAny&);                // disabled
                ListRelAny& operator= (const ListRelAny&);                 // disabled
   protected:   ListRelImpl* _impl;                                        // the implementation object
                List<Ref_Any> privGet () const;                            // get target list
                Ref_Any     privRetrieveAt (unsigned long pos) const;      // get indexed target
                void        privSet (const List<Ref_Any>& targets);        // assign target list
                void        privInsertBefore (const Ref_Any& target, unsigned long pos);
                void        privRemoveAt (unsigned long pos);              // remove from target
                void        privReplaceAt (const Ref_Any& target, unsigned long pos);
                int         privContains (const Ref_Any& target) const;    // check containment
                int         privFind (const Ref_Any& target, unsigned long& pos) const;
                Iterator<Ref_Any> privIterator () const;                   // iterate over target list
```

```
public:     ListRelImpl*        impl () const;                          // return implementation
                                ListRelAny (ListRelImpl* impl);
                                ~ListRelAny ();
                                operator List<Ref_Any> () const;
            int                 is_empty () const;
            unsigned long       cardinality () const;
            void                clear ();
            void                remove_all (); };
```

SimpleRel<T>, SetRel<T>, ListRel<T>

Die Templates SimpleRel<T>, SetRel<T> und ListRel<T> stellen typsichere Hüllen für die ungetypten Zugriffsklassen SimpleRelAny, SetRelAny und ListRelAny zur Verfügung. Sämtliche Methoden sind als Inline-Funktionen definiert. Wegen der unklaren Semantik bei direkter Zuweisung von Relationships werden einige der im Standard aufgeführten Konstruktoren und Operatoren nicht definiert. Man beachte, daß im Standard keinerlei Vergleichsoperatoren für mengenwertige und listenwertige Relationships definiert sind.

```
template <class T> class SimpleRel : public SimpleRelAny {
  public:                   SimpleRel (SimpleRelImpl* impl);
                            ~SimpleRel ();
            Ref<T>          get () const;
                            operator Ref<T> () const;
            T*              operator–> () const;
            T&              operator* () const;
            Ref<T>          operator() () const;
            SimpleRel&      set (const Ref<T>& r);
            SimpleRel&      set (T* r);
            SimpleRel&      set (T& r) ;
            SimpleRel&      operator= (const Ref<T>& r);
            SimpleRel&      operator= (T* r);
            SimpleRel&      operator= (T& r);
            void            operator() (const Ref<T>& r); };

template <class T> class SetRel : public SetRelAny {
  public:                   SetRel (SetRelImpl* impl);
                            ~SetRel ();
            Set< Ref<T> >   get () const;
            Set< Ref<T> >   operator() () const;
            Iterator< Ref<T> > create_iterator () const;
            int             contains_element (const Ref<T>& target) const;
            void            insert_element (const Ref<T>& targ;
            SetRel&         set (const Set< Ref<T> >& targets);
            SetRel&         operator= (const Set< Ref<T> >& targets);
            void            operator() (const Set< Ref<T> >& targets);
            void            remove_element (const Ref<T>& target); };

template <class T> class ListRel : public ListRelAny {
  public:                   ListRel (ListRelImpl* impl);
                            ~ListRel ();
            List< Ref<T> >  get () const;
            List< Ref<T> >  operator() () const;
            Ref<T>          retrieve_element_at (unsigned long pos) const;
            Ref<T>          operator[] (int position) const;
            Ref<T>          retrieve_first_element () const;
            Ref<T>          retrieve_last_element () const;
            Iterator< Ref<T> > create_iterator () const;
            int             contains_element (const Ref<T>& target) const;
            int             find_element (const Ref<T>& target, unsigned long& pos) cons;
            void            insert_element (const Ref<T>& targ;
            void            insert_element_after (const Ref<T>& target, unsigned long pos);
            void            insert_element_before (const Ref<T>& target, unsigned long pos);
            void            insert_element_first (const Ref<T>& target);
            void            insert_element_last (const Ref<T>& target);
            void            insert_element (const Ref<T>& target);
            ListRel&        set (const List< Ref<T> >& targets);
            ListRel&        operator= (const List< Ref<T> >& targets);
            void            operator() (const List< Ref<T> >& targets);
            void            replace_element_at (const Ref<T>& target, unsigned long pos);
            void            remove_element_at (unsigned long pos);
```

```
        void            remove_first_element ();
        void            remove_last_element ();
        void            remove_element (const Ref<T>& target); };
```

Implementierungsklassen

RelPathImpl

RelPathImpl stellt die gemeinsame Basisklasse für alle Pfadimplementierungen zur Verfügung. Es speichert eine Referenz auf das eigene Trägerobjekt und einen Funktionszeiger zur Ermittlung des inversen Pfadobjekts. Hierzu wird der Funktionszeigertyp RelAccFunctype definiert, der zu einem gegebenen Pobject ein bestimmtes zugehöriges RelPathImpl-Objekt zurückliefert. Weiterhin definiert RelPathImpl die Elementaroperationen simpleInsert und simpleRemove, die mit Hilfe von invSimpleInsert und invSimpleRemove auch direkt auf den inversen Pfadobjekten aufgerufen werden können. Mit getInvRelFor erhält man das inverse Pfadobjekt, und mark_modified führt ebendiese Operation auf dem eigenen Trägerobjekt aus. Genau wie bei den Zugriffsobjekten sind direkte Kopier- und Zuweisungsoperationen zwischen Relationships nicht verfügbar.

```
typedef RelPathImpl* (*RelAccFuncType)(const Ref_Any& hostObject);

class RelPathImpl {
    private:                    RelPathImpl (const RelPathImpl&);           // disabled
                RelPathImpl&    operator= (const RelPathImpl&);             // disabled
    protected:  Ref_Any         _myhost;                                    // the own Pobject
                RelAccFuncType  _invRelFunc;                                // retrieval function for inverse relationship
                                RelPathImpl (const Ref_Any& host, RelAccFuncType invRel=0);
                virtual         ~RelPathImpl ();
                void            mark_modified ();
                RelPathImpl*    getInvRelFor (const Ref_Any& target) const;
                void            invSimpleInsert (const Ref_Any& target);
                void            invSimpleRemove (const Ref_Any& target);
                virtual void    simpleInsert (const Ref_Any& target) =0;
                virtual void    simpleRemove (const Ref_Any& target) =0; };
```

SimpleRelImpl

SimpleRelImpl stellt die konkrete Implementierung für alle einwertigen Pfade zur Verfügung. Das referenzierte Objekt wird im Attribut _target abgelegt, das von den Elementaroperationen simpleInsert und simpleRemove manipuliert wird. Alle Funktionen sind geschützt (protected), da sie nur von der Zugriffsklasse und von Inversen benutzt werden dürfen. Für importierte und konvertierende Klassen werden weitere Subklassen zur Verfügung gestellt (siehe unten).

```
class SimpleRelImpl : public RelPathImpl {
    protected:  Ref_Any         _target;                                    // the referenced Pobject
                virtual void    simpleInsert (const Ref_Any& target);       // assign w/o check
                virtual void    simpleRemove (const Ref_Any& target);       // remove w/o check
                virtual void    check_target ();                            // reserved for derived cls
                virtual void    set (const Ref_Any& target);                // assign target
                virtual Ref_Any get () const;                               // return target
                virtual int     is_null () const;                           // test for target
                virtual void    clear ();                                   // remove target
    public:                     SimpleRelImpl (const Ref_Any& host, RelAccFuncType invRel);
                virtual         ~SimpleRelImpl (); };
```

SetRelImpl

SetRelImpl ist die Basisimplementierung für ungeordnete mengenwertige Relationships. Die Zielmenge wird in _targets abgelegt und wird durch simpleInsert und simpleRemove manipuliert.

```
class SetRelImpl : public RelPathImpl {
  protected:  Set<Ref_Any>              _targets;              // the referenced Pobjects
              const Set<Ref_Any>&       _get () const;
              virtual void              simpleInsert (const Ref_Any& target);
              virtual void              simpleRemove (const Ref_Any& target);
              virtual void              check_targets ();
              virtual void              insert (const Ref_Any& target);
              virtual void              set (const Set<Ref_Any>& targets);
              virtual Set<Ref_Any>      get () const;
              virtual int               is_empty () const;
              virtual int               contains (const Ref_Any& target) const;
              virtual unsigned long     cardinality () const;
              virtual Iterator<Ref_Any> iterator () const;
              virtual void              remove (const Ref_Any& target);
              virtual void              clear ();
  public:                               SetRelImpl (const Ref_Any& host, RelAccFuncType invRel);
              virtual                   ~SetRelImpl (); };
```

ListRelImpl

ListRelImpl stellt die Implementierung für geordnete mengenwertige Relationships, also Listen zur Verfügung. Die Zielliste wird in _targets abgelegt und wird durch simpleInsert, simpleRemove und die zusätzlichen positionsabhängigen Operationen simpleInsertBefore und simpleRemoveAt manipuliert.

```
class ListRelImpl : public RelPathImpl {
  protected:  List<Ref_Any>             _targets;              // the referenced Pobjects
              const List<Ref_Any>&      _get () const;
              virtual void              simpleInsert (const Ref_Any& target);
              virtual void              simpleRemove (const Ref_Any& target);
              virtual void              simpleInsertBefore (const Ref_Any& target, unsigned long pos);
              virtual void              simpleRemoveAt (unsigned long pos);
              virtual void              check_targets ();
              virtual void              insertBefore (const Ref_Any& target, unsigned long pos);
              virtual void              replaceAt (const Ref_Any& target, unsigned long pos);
              virtual void              set (const List<Ref_Any>& targets);
              virtual List<Ref_Any>     get () const;
              virtual Ref_Any           retrieveAt (unsigned long pos) const;
              virtual int               is_empty () const;
              virtual int               contains (const Ref_Any& target) const;
              virtual unsigned long     cardinality () const;
              virtual Iterator<Ref_Any> iterator () const;
              virtual int               find (const Ref_Any& target, unsigned long& pos) const;
              virtual void              removeAt (unsigned long pos);
              virtual void              clear ();
  public:                               ListRelImpl (const Ref_Any& host, RelAccFuncType invRel);
              virtual                   ~ListRelImpl (); };
```

SimpleRelImportImpl, SetRelImportImpl, ListRelImportImpl

Die drei Spezialisierungen für Importklassen stellen eine zusätzliche Initialisierungsfunktion für die Zielobjekte zur Verfügung und löschen die Referenzen im Destruktor wieder. Beides ist notwendig, um das Aktivieren bzw. Deaktivieren nicht als Update zu interpretieren.

```
class SimpleRelImportImpl : public SimpleRelImpl {
  public:             SimpleRelImportImpl (const Ref_Any& host, RelAccFuncType invRel);
              virtual ~SimpleRelImportImpl ();
              void    init (const Ref_Any& target); };

class SetRelImportImpl : public SetRelImpl {
  public:             SetRelImportImpl (const Ref_Any& host, RelAccFuncType invRel);
              virtual ~SetRelImportImpl ();
              void    init (const Set<Ref_Any>& targets); };

class ListRelImportImpl : public ListRelImpl {
  public:             ListRelImportImpl (const Ref_Any& host, RelAccFuncType invRel);
              virtual ~ListRelImportImpl ();
              void    init (const List<Ref_Any>& targets); };
```

SimpleRelDerivImpl, SetRelDerivImpl, ListRelDerivImpl

Die Implementierungen für Relationships zwischen Konvertierungsklassen sind als abstrakt definiert. Das heißt, daß für jede solche Relationship eine eigene Subklasse definiert werden muß, die zumindest die Funktion check_target bzw. check_targets überschreibt. Ausgehend von der Annahme, daß Aufdatierungen von abgeleiteten Relationships im allgemeinen nur schwer in geeignete Manipulationen der Basisklassen umgesetzt werden können, erzeugen die vordefinierten Zuweisungsoperationen entsprechende Fehlermeldungen. Steht für eine Relationship ein geeignetes Update-Verfahren zur Verfügung, können die Zuweisungsoperationen in der jeweiligen Subklasse überschrieben werden. Um eine ständige Neuberechnung der Relationship-Ziele zu vermeiden, wird das geerbte Attribut _target bzw. _targets zur Zwischenspeicherung verwendet. Vor jedem Zugriff wird dann check_target[s] aufgerufen, um sicherzustellen, daß dieses Attribut mit den aktuell gültigen Daten gefüllt ist. Beim ersten Aufruf wird folglich die entsprechende Herleitungsklausel ausgeführt, um das Ziel zu bestimmen; bei jedem weiteren Aufruf wird durch erneute Ausführung, Differenzbestimmung oder mittels Notifikationen das Ziel aktualisiert. Sind in der Sicht nur lesende Aktionen erlaubt, muß check_target[s] nach der Initialisierung überhaupt keine Aktionen mehr ausführen.

```
class SimpleRelDerivImpl : public SimpleRelImpl {
   protected:  virtual void      simpleInsert (const Ref_Any& target);
               virtual void      simpleRemove (const Ref_Any& target);
               virtual void      check_target ()=0;
               virtual void      set (const Ref_Any& target);
                                 SimpleRelDerivImpl (const Ref_Any& host, RelAccFuncType invRel);
               virtual           ~SimpleRelDerivImpl (); };

class SetRelDerivImpl : public SetRelImpl
   protected:  virtual void      simpleInsert (const Ref_Any& target);
               virtual void      simpleRemove (const Ref_Any& target);
               virtual void      check_targets ()=0;
               virtual void      set (const Set<Ref_Any>& targets);
               virtual void      insert (const Ref_Any& target);
               virtual void      remove (const Ref_Any& target);
                                 SetRelDerivImpl (const Ref_Any& host, RelAccFuncType invRel);
               virtual           ~SetRelDerivImpl (); };

class ListRelDerivImpl : public ListRelImpl {
   protected:  virtual void      simpleInsertBefore (const Ref_Any& target, unsigned long pos);
               virtual void      simpleRemoveAt (unsigned long pos);
               virtual void      check_targets ()=0;
               virtual void      insertBefore (const Ref_Any& target, unsigned long pos);
               virtual void      removeAt (unsigned long pos);
                                 ListRelDerivImpl (const Ref_Any& host);
               virtual           ~ListRelDerivImpl (); };
```

H.4 Data Dictionary

Der ODMG-Standard definiert kein eigenes Data Dictionary. In [Cat93] und [Cat94] sind vereinzelte Definitionen für Metaklassen zu finden, und in [LAC*93] ist ein Bild mit einem „vereinfachten Metaschema“ angegeben. Die einzelnen Teile passen jedoch nicht zusammen, Teile fehlen oder sind widersprüchlich und viele Funktionen sind nicht erklärt. Außerdem erhält man den Eindruck, daß Instanz- und Metaebene nicht sauber voneinander getrennt sind. Das von CORBA definierte Data Dictionary ([OMG91]) ist hingegen sehr rudimentär. Außer einer generischen Enthaltenseinsbeziehung gibt es kaum Schemainformationen, die Beziehungen sind immer ungetypt, und das TypeCode-Verfahren zur Datentypbeschreibung ist ebenfalls sehr implementierungsorientiert.

Aus diesem Grunde wurde in IRO–DB ein eigenes Metaschema entworfen. Der Entwurf von [LAC*93] wurde erweitert und teilweise umbenannt, um sowohl kompatibler mit dem CORBA-

Entwurf zu sein als auch die Unterscheidung zwischen Daten und Metadaten deutlicher hervorzuheben. Properties und Methoden sind nun getrennt modelliert, Beschreibungsklassen für Datentypen, Methodenparameter, Exceptions, Datenbankanfragen, Schemas und Datenbanken sind hinzugekommen. Die implementierungsspezifischen Klassen class und object entfallen.

Für den Entwurf des Data Dictionary betrachtet man die Spezifikationssprache ODL. Für eine Klassendefinition müssen demnach die folgenden Informationen im Data Dictionary abgelegt werden:

- der *Klassenname*
- die Liste der *Supertypen*
- optional der *Extent*, bestehend aus dem Extentnamen und einer Referenz auf die Instanzenmenge der Klasse, sowie Zugriffsmethoden zum Hinzufügen und Löschen von Instanzen
- eine Liste von *Schlüsselmengen*
- eine Liste von *Attributen*, bestehend aus Name und Datentyp
- eine Liste von *Relationshippfaden*, bestehend aus Name, Wertigkeit, Zielklasse und Verweis auf den zugehörigen inversen Pfad
- eine Liste von *Methoden* (Operationen) mit Name, Parameterliste und Exceptions
- eine Zuordnung von *Lese- und Schreibmethoden* zu einem Attribut, sofern spezifische Funktionen get_value und set_value verwendet werden sollen

Hinzu kommen die Informationen, die nicht an einzelne Klassen gebunden sind:

- Ein *Schema* faßt eine Menge von Klassen (Interfaces) zusammen und definiert benannte Anfragen, Datentypen und Exceptions.
- Eine *Datenbank* ist die Instanziierung eines Schemas. Hier findet die eigentliche Datenhaltung statt.
- Eine *Relationship* faßt jeweils zwei inverse Pfade zusammen.

Betrachtet man nur die prozedurale Darstellung meines Sichtsystems, so reicht dieses ODMG-Standardschema bereits zur Darstellung der Sichten aus. Die gesamte Abbildungsinformation zur Herleitung von Objekten und Daten ist in den Methoden get_extent, get_value und set_value repräsentiert. Will man jedoch auch Anfrageexpansion für optimierten deklarativen Zugriff zur Verfügung stellen, müssen die entsprechenden Herleitungsklauseln auch im Data Dictionary verfügbar sein. Unabhängig davon ist es aber immer sinnvoll, soviel verfügbare Schemainformation wie möglich im Data Dictionary aufzubewahren. Zusätzliche Informationen helfen bei der Interpretation des Schemas, erlauben die Wiedererzeugung der ursprünglichen Schemadefinition und helfen so zum Beispiel bei der Schemaevolution. Als sichtspezifische Informationen kommen also noch hinzu:

- ein Verweis auf das *orig-Attribut* von virtuellen Klassen (Das Attribut selbst ist als normales, privates Attribut modelliert, das vom Benutzer nicht zugreifbar ist.)
- eine Markierung *is_private*, um Attribute als privat zu kennzeichnen
- die *Kategorie* der Klasse (kind)
- die *Herleitungsklauseln* für Attribute, Pfade, Relationships, Extents und Methoden in Form von OQL-Anfragen
- der Name und die Speicheradresse von *Wrapperfunktionen* für Methoden und Konstruktoren

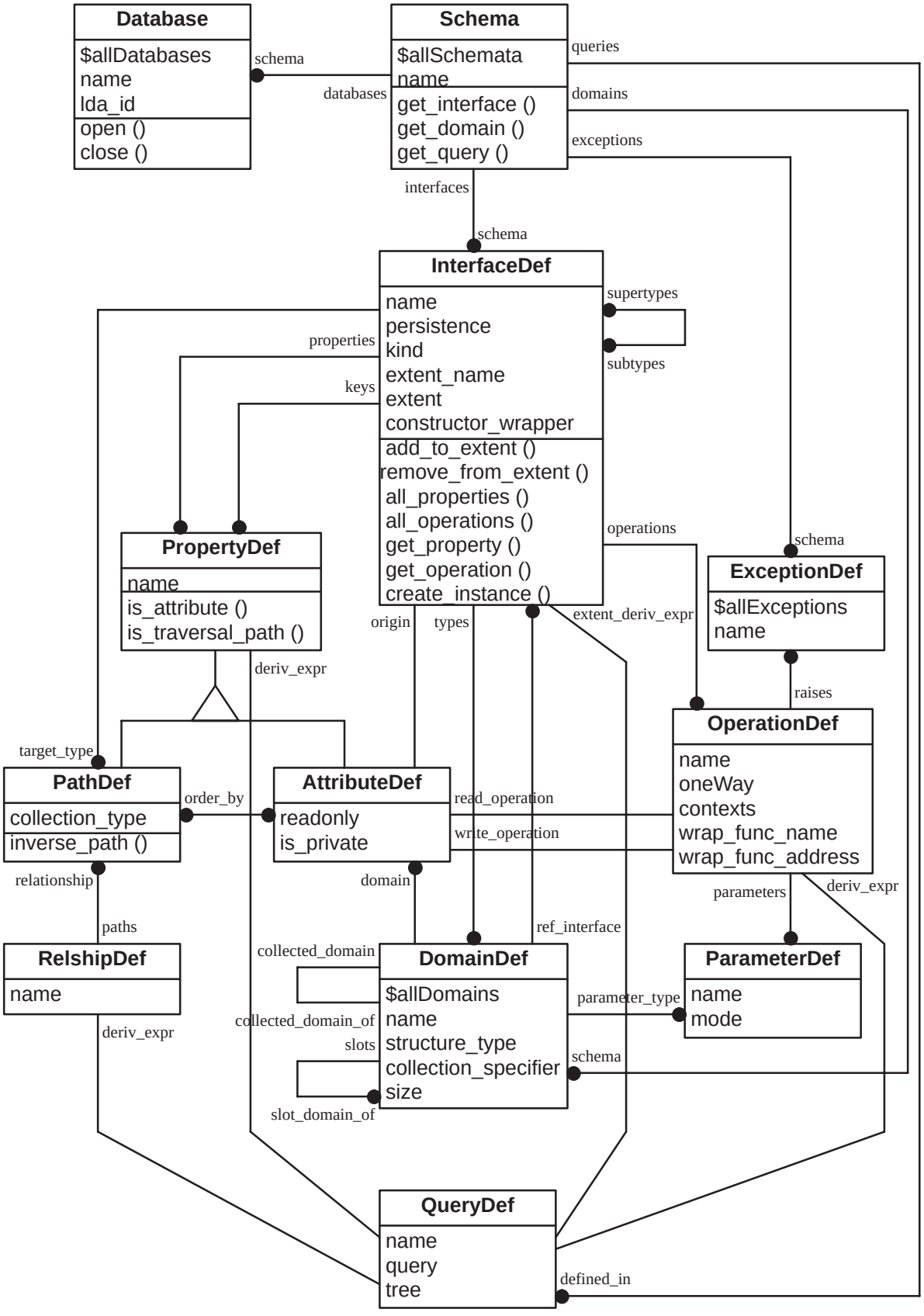


Bild 30: Das Data Dictionary

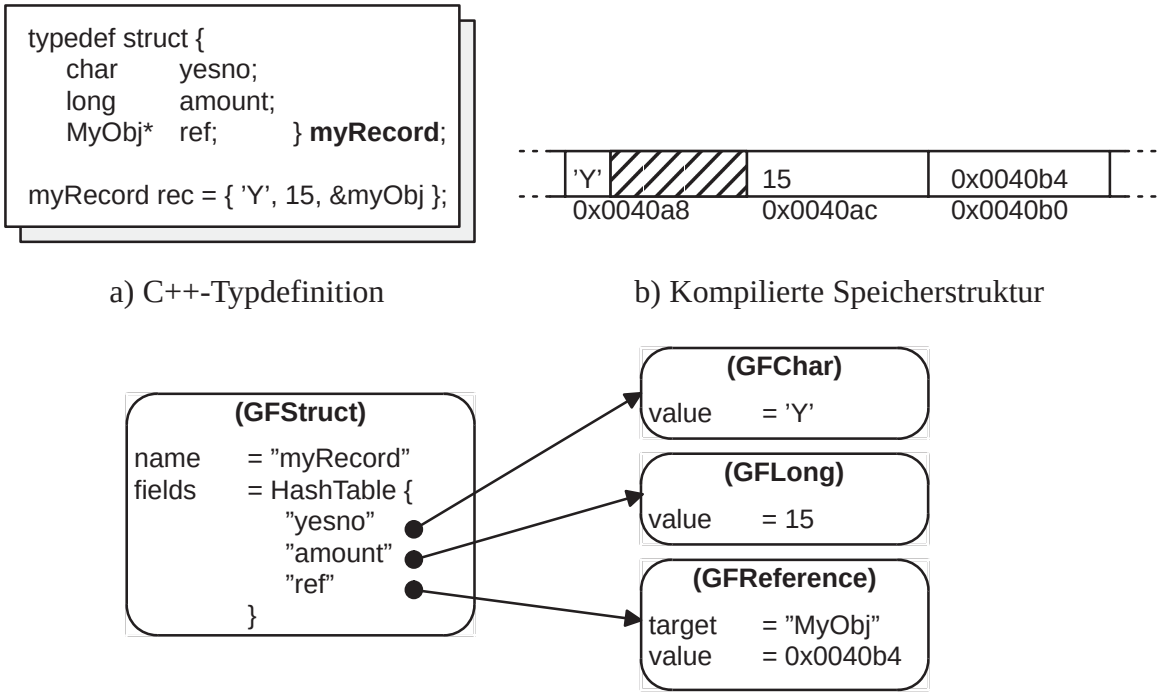
- weitere *Hilfsstrukturen* und *Hashtabellen*, um den Zugriff auf das Data Dictionary zu beschleunigen

Bild 30 zeigt das von uns in IRO–DB verwendete Data Dictionary in OMT-Notation. Als Implementierung dient die C++-Sprachbindung der entsprechenden ODL-Spezifikation. Weiterhin gehört zum Data Dictionary eine Zugriffsklasse IDD, die die allgemeine Grundfunktionalität des Data Dictionary, wie etwa Datenbankimport, Schema-Übersetzung usw., zur Verfügung stellt. Das leere Data Dictionary enthält bereits die Beschreibungen für die Systemklassen Pobject, ImportClass und DerivedClass. Für die Definition der Klasse RelshipDef gibt es zwei Motivationen. Zum einen entspricht die Gruppierung von zwei Pfaden zu einer Relationship der ODMG-Interpretation. Zum anderen erlaubt diese Modellierung, Relationships über eine eigene Herleitungsklausel zu definieren und die beiden Zugriffspfade durch Selektionen auf diesem Ausdruck darzustellen. Dieser Ansatz ist zeitweise im IRO–DB-System verfolgt worden, da er eine sauberere Modellierung mit besserer Konsistenzkontrolle zuläßt, als die direkte Berechnung der Zielobjekte aus den Pfadausdrücken. Ersetzt man jedoch die direkte Zielobjekterzeugung durch eine Selektion auf der Zielklasse, kann man mit geringerem Spezifikationsaufwand das gleiche Ziel erreichen. Bei diesem letztendlich gewählten Verfahren ist die angegebene RelshipDef-Klasse nicht mehr notwendig. Die umfassende Spezifikation des Data Dictionary mit allen Schnittstellen, Klassendefinitionen und Diskussionen ist im IRO–DB-Dokument [D4-3/1] gegeben.

H.5 Das generische Datenformat GF

In Abschnitt 4.5.2 wurde auf die Verwendung eines generischen Datenformats für die Anfrageauswertung und generische Parameterübergabe hingewiesen. Dieses *Generische Datenformat* (kurz *GF*) besteht aus einer vordefinierten Menge von Klassen, mit denen jeder Datentyp modelliert werden kann. Für die Basisdatentypen gibt es die Klassen GFInteger, GFChar, GFString[53] usw., die jeweils einen Wert dieses Typs aufnehmen können. Strukturen werden durch eine GFStruct-Klasse repräsentiert, die intern eine Menge benannter Felder verwaltet, zum Beispiel in Form einer Hashtabelle. Entsprechende Klassen gibt es auch für die anderen Typkonstruktoren (GFSet, GFList, ...) sowie für Objektreferenzen (GFReference) und für die Verallgemeinerung aller Typen (GFAny). Obwohl die Klassen selbst unveränderlich sind, können mit ihnen alle Datenstrukturen modelliert werden. Ein Beispiel für eine Struktur mit ihrer Darstellung als kompilierter C++-Wert und im generischen Datenformat ist in Bild 31 angegeben. Tabelle 10 enthält die vollständige Abbildungsdefinition zwischen ODL-Datentypen und GF-Typen des generischen Datenformats. Auf alle Daten wird grundsätzlich über ein GFAny-Objekt zugegriffen. Die Einträge in der Spalte GF(T) sollen nur die Schachtelung der verwendeten Typobjekte verdeutlichen; es handelt sich nur teilweise um echte Templates. Für Objekte gibt es im generischen Datenformat keine Entsprechung, da auf sie nur über Referenzen zugegriffen wird. Attributzugriffe liefern den jeweiligen Datentyp des Attributs zurück, Relationshipzugriffe liefern Referenzen bzw. Mengen oder Listen von Referenzen. Details zu den Datentypen und der Abbildung findet man in [Leg96] und [Bus96]. Die generischen Datenstrukturen sind von Régis Legoff für die Kommunikationsschicht entworfen und implementiert worden. Einige Rohentwürfe, sowie die Anwendung der Strukturen für die Anfrageauswertung und die Wrapperfunktionen stammen von mir.

53. Die Namensgebung im IRO-DB-System unterscheidet sich leicht von den hier angegebenen Bezeichnungen.



a) C++-Typdefinition b) Kompilierte Speicherstruktur

c) Darstellung im generischen Datenformat

Bild 31: C++-Format und generisches Datenformat einer Beispielstruktur

	ODL-Datentyp T	**Generischer Datentyp GF(T)**
Atomare Literale	short	GFAny<GFShort>
	long	GFAny<GFLong>
	unsigned short	GFAny<GFShort>
	unsigned long	GFAny<GFLong>
	float	GFAny<GFFloat>
	double	GFAny<GFDouble>
	char	GFAny<GFChar>
	boolean	GFAny<GFInteger>
	octet	*nicht verfügbar*
	any	GFAny
	void	void
Strukturierte Literale	array<T>, array<T,n>	GFAny<GFArray<GF(T)>>
	sequence<T>, sequence<T,n>	*nicht verfügbar*
	set<T>	GFAny<GFSet<GF(T)>>
	list<T>	GFAny<GFList<GF(T)>>
	bag<T>	GFAny<GFBag<GF(T)>>
	string, string<n>	GFAny<GFString>
	enum e { $name_1$, ... }	*nicht verfügbar*
	struct s { T_1 $name_1$; ...; }	GFAny<GFStruct< $name_1 \rightarrow GF(T_1)$; ...;>>
	union u switch (T_S) { ...; }	*nicht verfügbar*
Arrays fester Länge	T[n]	GFAny<GFArray<GF(T)>>
Referenzen	OT	GFAny<GFReference<OT>>

Tabelle 10: GF(T) – Abbildung von ODL-Datentypen auf GF-Typen

H.6 Hilfsfunktionen und Wrapper

Im folgenden werden die einzelnen durch den Präprozessor erzeugten Funktionen definiert, die das Zusammenspiel von Anfragebearbeitung, Objektverwaltung und Sprachbindung ermöglichen (siehe Kapitel 4.5.3). Die vollständige Spezifikation des für ein Schema zu erzeugenden Codes ist in [Bus96] definiert. Neben Details zu den hier beschriebenen Funktionen sind auch weitere Konstruktoren, Destruktoren, Operatoren und der Gesamtaufbau der C++-Header- und Implementierungsdatei beschrieben.

H.6.1 Parameterübergabe an das Anfragesystem

Das generische Datenformat wird nur innerhalb des Anfragesystems verwendet; sowohl die Anwendungsprogramme als auch die verwendete direkte Sprachbindung arbeiten mit kompilierten C++-Strukturen. Um Parameter an eine Anfrage zu übergeben, oder Daten zwischen der Datenbank und dem Anfragesystem auszutauschen, muß jede C++-Struktur in die entsprechende GF-Struktur umgewandelt werden können. Hierzu generiert der Präprozessor für jeden im Schema vorhandenen Datentyp eine Konvertierungsfunktion. Den Polymorphismus von C++ ausnutzend hat jede Funktion den Namen getGF, nimmt einen Wert vom entsprechenden Typ als Parameter und liefert die äquivalente GF-Darstellung als GFAny zurück:

```
GFAny* getGF (CPP(IN,T) inp);
```

Eine Grundmenge von Funktionen für die Basisdatentypen, einschließlich der Objektreferenzen, wird dabei in einer Funktionsbibliothek zur Verfügung gestellt. Die Umwandlungsfunktionen für Mengen- und Listentypen können als Templates realisiert werden; der C++-Compiler erzeugt dann die konkreten Spezialisierungen. Die einzigen neu zu generierenden Funktionen betreffen die schemaabhängigen Strukturen und Varianten. Hierbei werden die einzelnen Komponenten rekursiv mit den passenden getGF-Funktionen umgewandelt und die Ergebnisse dann kombiniert. Für die in Bild 31 definierte Struktur wird zum Beispiel der folgende Code erzeugt:

```
GFAny* getGF (const precord& inp) {
    GFStruct* result = new GFStruct;
    result→insert_field ("yesno", getGF(inp.yesno));
    result→insert_field ("amount", getGF(inp.amount));
    result→insert_field ("ref", getGF(inp.ref));
    return (GFAny*) result; };
```

H.6.2 Ergebnisrückgabe vom Anfragesystem

Die umgekehrte Situation ergibt sich bei der Rückgabe der Ergebnisse vom Anfragesystem an das Anwendungsprogramm sowie bei der Übergabe von Parametern bei Methodenaufrufen durch das Anfragesystem. Für jede getGF-Funktion wird also eine Inverse assignGF definiert, die ein gegebenes GFAny in die äquivalente C++-Struktur überführt. Um auch hierbei Polymorphismus ausnutzen zu können, muß der Zieltyp aber in der Parameterliste auftauchen, da sich die Funktionen ansonsten nur im Rückgabetyp unterschieden. Wenn sich das übergebene GFAny-Argument nicht in den Zieltyp konvertieren läßt, wird ein Ausnahmesignal erzeugt bzw. ein Fehlerstatus zurückgeliefert. Dieser Fall konnte bei den getGF-Funktionen nicht auftreten, da das zurückgegebene GF-Format jeden beliebigen Eingabewert darstellen kann.

```
void assignGF (CPP(OUT,T) target, GFAny* inp);
```

Das Implementierungsschema entspricht dem der getGF-Funktionen. Funktionen für die Basisdatentypen und die Collections werden in einer Bibliothek und als Templates zur Verfügung ge-

stellt. Für Strukturen und Varianten werden Funktionen generiert, die rekursiv die einzelnen Felder zuweisen. Hier ist die assignGF-Funktion für unsere Beispielstruktur:

```
void assignGF (precord& target, GFAny* inp) {
    if ( ! inp || ! inp→isStruct() ) throw GFBadFormat();
    GFStruct* s = (GFStruct*) inp;
    if ( s→cardinality() != 3 ) throw GFBadFormat();
    assignGF (target.yesno, s→get_field("yesno"));
    assignGF (target.amount, s→get_field("amount"));
    assignGF (target.ref, s→get_field("ref")); };
```

H.6.3 Zugriff auf Objektfelder

Mittels getGF und assignGF können Datenstrukturen zwischen Anwendungsprogrammen und dem Anfragesystem ausgetauscht werden. Datenstrukturen in C++-Format werden bei der Parameterübergabe in das GF-Format umgewandelt, dann während der Auswertung in diesem generischen Format bearbeitet, und schließlich bei der Ergebnisrückgabe wieder in C++-Format zurückgewandelt. Während der Auswertung erfolgen also keine Zugriffe auf die ursprünglichen C++-Variablen. Dieses Verfahren funktioniert jedoch nur für literale Datenwerte und nicht für Objekte. Objektreferenzen können zwar im GF-Format übergeben werden, bei der Dereferenzierung muß aber stets zum ursprünglichen Objekt aufgelöst werden, da Methodenaufrufe die C++-Speicherstruktur mit Datenfeldern und virtuellen Methodentabellen erwarten. Um für die Anfrageauswertung generisch auf die Datenfelder eines Objekts zugreifen zu können, müssen wiederum geeignete Lese- und Schreibmethoden generiert werden. Zu jeder Klasse werden die Methoden priv_get_property und priv_set_property hinzugefügt. priv_get_property erhält als Parameter den gesuchten Attributnamen und liefert den zugehörigen Wert im GF-Format zurück. Für die Umwandlung geerbter Datenfelder wird rekursiv die priv_get_property-Funktion der Superklasse aufgerufen. Ein gefundener Attributwert kann mit den bereits definierten getGF-Methoden umgewandelt werden; existiert das gesuchte Attribut nicht, wird ein Fehlersignal erzeugt. Bei Mehrfachvererbung müssen die in der Rekursion erzeugten Fehlersignale abgefangen und nur dann weitergereicht werden, wenn das Attribut in keinem der alternativen Pfade gefunden wurde. Der zu generierende Code ist wiederum sehr einfach: sowohl der Zugriff auf das Attribut als auch die Auswahl der korrekten getGF-Funktion wird vom Compiler übernommen. Die Auswahl des Attributs erfolgt durch eine Fallunterscheidung. Für den Schreibzugriff auf ein Attribut existiert die inverse Funktion priv_set_property. Sie erhält den Attributnamen und den zuzuweisenden Wert im GF-Format und verwendet dann assignGF für die eigentliche Zuweisung. Das folgende Codefragment zeigt eine sehr einfache Implementierung dieser Funktionen für die importierte Personenklasse aus Beispiel 3.4:

```
GFAny ADRDB_Person::priv_get_property (const char* prop) const {
    if ( strcmp (prop, "lname") == 0 )          return getGF (lname);
    else if ( strcmp (prop, "fname") == 0 )     return getGF (fname);
    else if ( strcmp (prop, "birthdate") == 0 ) return getGF (birthdate);
    else if ( strcmp (prop, "address") == 0 )   return getGF (address);
    else return ImportClass::priv_get_property (prop); };

void ADRDB_Person::priv_set_property (const char* prop, GFAny val) {
    if ( strcmp (prop, "lname") == 0 )          assignGF(lname,val);
    else if ( strcmp (prop, "fname") == 0 )     assignGF(fname,val);
    else if ( strcmp (prop, "birthdate") == 0 ) assignGF(birthdate,val);
    else if ( strcmp (prop, "address") == 0 )   assignGF(address,val);
    else ImportClass::priv_set_property (prop,val); };
```

In diesem Beispiel wird direkt auf die Datenfelder zugegriffen. Statt dessen könnte man auch die entsprechenden Lese- und Schreibmethoden für die Datenfelder verwenden. Weiterhin verwendet das obige Beispiel sehr aufwendige und langsame Zeichenkettenvergleiche, um das gesuchte Datenfeld zu identifizieren. Eine deutliche Effizienzsteigerung erhält man durch Umwandlung

der Zeichenketten in numerische Konstanten. Verwendet man etwa eine globale Hashtabelle, in der zu Beginn alle erlaubten Zeichenketten mit ihren zugeordneten Konstanten registriert werden, so muß jede Zeichenkette innerhalb einer Anfrage nur einmal untersucht werden. Alle weiteren Zugriffe auf die Objekte können mit einem effizienten switch-Kommando ausgeführt werden. Problematisch ist bei diesem Verfahren allerdings, daß die verwendeten Konstanten global, oder zumindest innerhalb der kompletten Vererbungshierarchie einer Klasse eindeutig sein müssen. Die obige Lesemethode sieht dann wie folgt aus:

```
GFAny ADRDB_Person::priv_get_property (PropertyIdType prop) const {
    switch (prop) {  case Part_lname_id:      return getGF (lname);
                     case Part_fname_id:      return getGF (fname);
                     case Part_birthdate_id:  return getGF (birthdate);
                     case Part_address_id:    return getGF (address);
                     default:                 return ImportClass::priv_get_property (prop); } };
```

H.6.4 Methodenaufrufe

In OQL ist es erlaubt, Methoden innerhalb einer Anfrage aufzurufen. Um zur Laufzeit die Probleme bei der Bestimmung der korrekten Methode und der Parameterübergabe zu lösen, wird für jede Methode eine zusätzliche Wrapperfunktion erzeugt. Sie übernimmt die Parameter im GF-Format, wandelt sie in flaches C++-Format, ruft die eigentliche Funktion auf und wandelt das Ergebnis und eventuelle Ausgabeparameter wieder in GF-Format zurück. Das folgende Codefragment zeigt den Aufbau eines solchen Wrappers:

```
extern ”C” GFAny wrappername (Ref_Any obj, GFStruct* args) {
    Type1 att1; Type2 att2; Type3 res;
    assignGF (att1, args→get_field(”att1”));      // convert input parameters
    assignGF (att2, args→get_field(”att2”));
    res = ((Class*)obj)→method (att1, att2);      // call the method
    return getGF (res); };                        // convert result and output parameters
```

Beim direkten Methodenaufruf im Inneren der Funktion wird durch den vom Compiler generierten Code automatisch die spezifischste Methode aufgerufen. Das Anfragesystem muß nur noch die statische Methodendefinition in dieser Klasse kennen und muß sich nicht um Redefinitionen in Subklassen kümmern — die Ermittlung der korrekten statischen Funktion für eine gegebene Parameterliste bleibt allerdings Aufgabe des Anfragesystems. Alle Wrapper haben die gleiche Signatur und verwenden das GF-Format zur Datenübergabe; die Signaturvielfalt und die Konvertierung in flache C++-Strukturen sind im Inneren der Wrapper versteckt. Die Namen der Wrapper werden nach einem eigenen Schema vergeben und durch die Deklaration als „extern ”C”“ vom Compiler unverändert übernommen (siehe [Bus96]). Das Auffinden der Startadresse mittels dlsym ist also auch kein Problem mehr.

Anmerkung: Wenn auch für alle Lese- und Schreibzugriffe auf Attribute eine eigene Wrapperfunktion erzeugt wird, können die Funktionen priv_get_property und priv_set_property entfallen, da die Datenfelder durch die dynamisch aufrufbaren Funktionen direkt adressierbar sind. Dabei entfallen dann auch die zeitaufwendigen Zeichenkettenvergleiche bzw. die Definition globaler Identifikatoren.

H.6.5 Objekterzeugung

Neben der Arbeit mit existierenden Daten erlaubt OQL auch, neue Daten und Objekte zu erzeugen. Die Erzeugung neuer Daten stellt dabei kein Problem dar, da beliebige Strukturen im GF-Format dargestellt werden können. Zu beachten ist hier nur, daß für das Endergebnis der Anfrage eine entsprechende assignGF-Funktion vorhanden sein muß. Bei Objekten hingegen muß zur

Neuerzeugung eine Methode aufgerufen werden, die das Objekt in der Datenbank anlegt und schließlich durch einen Konstruktoraufruf als C++-Objekt zur Verfügung stellt. Diese create_object-Methode hat für alle Klassen dieselbe Signatur, da die Struktur mit den initialen Attributwerten immer als GF-Struktur (ALHolder) übergeben werden kann. In der Methode werden gegebenenfalls Schlüsselbedingungen überprüft, dann wird ein Objekt mit den geforderten Attributwerten angelegt und zurückgegeben. Auch für diese Methode wird eine Wrapperfunktion erzeugt, die dynamisch vom Anfragesystem aufgerufen werden kann. Bei diesen Wrappern entfällt die Übergabe eines Zielobjekts, und die Konvertierung der Argumente nach C++ ist ebenfalls überflüssig, da create_object selbst mit dem GF-Format umgehen kann. Funktion und Wrapper für die Importklasse ADRDB_Person sehen wie folgt aus:

```
Ref_Any ADRDB_Person::create_object (ALHolder& props) {
    assertCreatePerm(_metaClass);                    // check permission with security system
    // ...                                           // check key information etc.
    return theOM→createObject (_metaClass, props); };   // create the object

extern ”C” Ref_Any wrap_ADRDB_Person__create_object_ALHolder_ (ALHolder& props) {
    return ADRDB_Person::create_object (props); };
```

Die Implementierung von create_object unterscheidet sich für die verschiedenen Kategorien. Bei normalen Klassen kann einfach ein Konstruktor aufgerufen werden. Für Importklassen stellt der Objektmanager eine eigene createObject-Methode zur Verfügung, die die zur Objekterzeugung notwendige OQL-Anfrage an die externe Datenbank schickt und die gelieferte Ergebnis-OID als Referenz zurückliefert. Für Konvertierungsklassen müssen in den zugrunde liegenden Basisklassen neue Instanzen angelegt werden, von denen das neue Konvertierungsobjekt abgeleitet wird. Da jede Objekterzeugung klassenspezifisch ist, muß die create_object-Methode vom Entwickler zur Verfügung gestellt werden. Die Wrapper werden automatisch generiert.

create_object sollte nur vom Anfragesystem verwendet werden, da das GF-Format für Anwendungsprogramme unbekannt ist. Will man dem Benutzer bzw. dem Anwendungsprogramm ebenfalls die Möglichkeit geben, neue Objekte zu erzeugen, so sollten hierfür zusätzliche Konstruktoren mit geeigneten Parameterlisten implementiert werden. Existiert ein solcher „Anwendungskonstruktor“ mit vollständiger Parameterliste, kann auch dieser vom Wrapper aus aufgerufen werden. Die Umwandlung vom GF-Format in C++-Format erfolgt dann im Wrapper.

H.6.6 Objektaktivierung

Neben der Erzeugung neuer Objekte muß auch das Aktivieren existierender Datenbankobjekte unterstützt werden. Hierzu wird ein Konstruktor erzeugt, der die zur Aktivierung notwendige Parameterliste übergeben bekommt. Für die Übergabe wird die private Klasse Activator verwendet, die intern eine GF-Struktur mit den Werten aller Datenfelder und die zur Objektidentifikation notwendigen Referenzinformationen enthält. Hierdurch wird der Aktivierungskonstruktor von anderen Konstruktoren unterscheidbar, und es wird verhindert, daß der Konstruktor von Anwendungsprogrammen aufgerufen wird. Nach der Aktivierung wird das Objekt im aktiven Extent seiner Klasse, also der Liste der im Hauptspeicher befindlichen Instanzen, eingetragen. Bei der Initialisierung der Datenfelder ist zu beachten, daß man nicht die Schreibmethoden der Datenfelder verwendet. Diese würden die Zuweisung als Update betrachten, mark_modified aufrufen, die ‚neuen‘ Werte in die Datenbank schreiben und bei Relationships auch die Inverse aufdatieren. Für den Aufruf des Aktivierungskonstruktors existiert ebenfalls eine Wrapperfunktion. Sie erhält als zusätzlichen Parameter die Datenbank, in der das Objekt aktiviert werden soll, um damit gemäß ODMG–93 den new-Operator aufzurufen.

```
ADRDB_Person::ADRDB_Person (Activator& act)
    : ImportClass (act), priv_lname (0), priv_fname (0), priv_birthdate (0)
    , address (new SimpleRelImportImpl (this, ADRDB_Address::priv_getrel_residents)) {
    // ...                              // additional initialization steps
    add_to_extent(); };                 // register object with the class extent

class Activator {
    friend class OM; friend class DOM; friend class ImpRefHdlr; friend class Pobject;
    private:    Ref_Any     selfref;
                GFStruct*       props;
                Activator (Ref_Any r, GFStruct* p);
                ~Activator (); };

extern "C" Pobject* wrap_ADRDB_Person__ADRDB_Person_Activator_ (Database* db, Activator& act)
    { return new (db, "ADRDB_Person") ADRDB_Person (act); }
```

Man beachte, daß die Anfrageauswertung zwischen den Aktivierungs- und Erzeugungskonstruktoren unterscheiden muß. Kommt in einer herkömmlichen OQL-Anfrage ein Konstruktor-Aufruf vor, so handelt es sich stets um eine Objekterzeugung. In meinem Sichtsystem sind jedoch auch die Extentklauseln mit Hilfe von Konstruktoren definiert. Diese sind aber stets als Objektaktivierung zu interpretieren, da sie nur einen Zugriffsmechanismus auf (real oder virtuell) in der Datenbank bereits existierende Objekte darstellen. Alle Konstruktoraufrufe, die bereits in einer Benutzeranfrage auftreten, müssen demnach als Erzeugungskonstruktoren markiert werden, während die durch Expansion der Extentklauseln eingefügten Konstruktoren als Aktivierungskonstruktoren zu markieren sind (siehe Expansionsbeispiel auf Seite 58).

H.6.7 Deaktivierung und Löschen

Zum Entfernen eines Objekts aus dem Speicher werden zwei weitere Methoden generiert. Der Destruktor entfernt das Objekt aus dem Extent und gibt alle belegten Ressourcen frei. Vor dem Aufruf des Destruktors muß die private Funktion priv_readout aufgerufen werden, die alle aktuellen Attributwerte in eine GF-Struktur überträgt, damit sie vom Objektmanager in die Datenbank zurückgeschrieben werden können. (Schreib-Caching, siehe Abschnitt 4.2.5.2 und Kapitel 5.)

```
ADRDB_Person::~ADRDB_Person () {
    remove_from_extent ();
    REMOVE_STRING (priv_lname);
    REMOVE_STRING (priv_fname);
    REMOVE_STRING (priv_birthdate); };

GFStruct* ADRDB_Person::priv_readout () const {
    GFStruct* values;
    values = ImportClass::priv_readout();
    values→insert_field ("lname", getGF(lname()));
    values→insert_field ("fname", getGF(fname()));
    values→insert_field ("birthdate", getGF(birthdate()));
    values→insert_field ("address", getGF(address()));
    return values; };
```

Der Destruktor kümmert sich nur um das Deaktivieren des Objekts. Für das dauerhafte Löschen eines Objekts aus der Datenbank sind zuvor andere Operationen notwendig. Bei normalen Klassen ist der Destruktoraufruf gleichbedeutend mit einer Objektlöschung, da keine Aktivierungsmechanismen unterstützt werden. Bei Importobjekten übernimmt die Objektverwaltung selbst den Löschvorgang, indem sie ein entsprechendes Kommando an die externe Datenbank absetzt. Bei Konvertierungsklassen hingegen ist die Objektlöschung, genau wie die Objekterzeugung, für jede einzelne Klasse separat zu implementieren. Hierzu dient die Methode priv_destroy, die die notwendigen Operationen durchführt.

```
void Author::priv_destroy () {
    priv_orig.person.delete_object ();
    priv_orig.author.delete_object (); };
```

H.6.8 Metainformation

Die letzte Hilfsfunktion, die vom Präprozessor erzeugt wird, ist die Füllroutine für das Data Dictionary. Wie in Kapitel 4.5.1 erläutert, wird für jedes Schema eine spezielle Funktion [schemaname]_init_IDD erzeugt, mit der das Data Dictionary dynamisch zur Laufzeit gefüllt wird. Diese Schemainitialisierungsroutine ist wie die Wrapperfunktionen als freie Funktion mit „C-Linkage“ definiert, um sie beim Öffnen der Datenbank dynamisch aufrufen zu können. Hierdurch kann eine Anwendung mit verschiedenen Datenbanken arbeiten. Wird stets nur mit einem einzigen Schema gearbeitet, kann man auf das Codieren des Schemanamens in den Funktionsnamen verzichten und die Initialisierungsroutine fest im Hauptprogramm aufrufen. Der folgende Programmcode zeigt als Beispiel den Anfang einer solchen Initialisierungsmethode:

```
Schema* _IDD_exSchema = 0;                          // public schema object

extern ”C” Database* exSchema_init_IDD () {
    Database* db = new Database (”exSchema”);       // create database object
    _IDD_exSchema = db→createSchema (”exSchema”);   // create schema object
    //————————————————————————————————
    // Interface ADRDB_Person
    ADRDB_Person::_metaClass = _IDD_exSchema→createInterface (”ADRDB_Person”);
    ADRDB_Person::_metaClass→setMetaInfo (imported, persistent, extent, ...);
    ADRDB_Person::_metaClass→constructor_wrapper = wrap_ADRDB_Person__ADRDB_Person_Activator_;
    ADRDB_Person::_metaClass→addAttribute (”lname”, ”String”, isKey);
    ADRDB_Person::_metaClass→addAttribute (”fname”, ”String”, isKey);
    ADRDB_Person::_metaClass→addAttribute (”birthdate”, ”String”, noKey);
    ADRDB_Person::_metaClass→addRelationship (”address”, ”ADRDB_Address::residents”, MANY_TO_ONE);

    //————————————————————————————————
    // Interface ADRDB_Address ...
    // Interface Person ...
    // Interface Address ...
    return db; }
```

H.7 Weitere Klassen

Einige wichtige Klassen sind bisher noch nicht besprochen worden. Hierzu gehören die drei Datenbank-Basisklassen Pobject, ImportClass und DerivedClass als Wurzelklassen für alle Datenbankobjekte, die Systemklassen IRO, Database und Transaction, sowie die Funktionen zur Auswertung von OQL-Anfragen. Details hierzu findet man im IRO–DB-Dokument [D5-1/2]. Neben diesen Klassen definiert die Objektverwaltung eine ganze Reihe an Hilfsfunktionen, die zum Beispiel die Zugriffe auf andere Module vereinfachen und beschleunigen. Speziell für den Zugriff auf das Data Dictionary werden Hashtabellen angelegt, um eine schnelle Datensuche zu gewährleisten. Details zu diesen Funktionen, sowie zu den oben beschriebenen Klassen, findet man in [D4-2/3].

Basisklassen

Pobject, Persistent_Object

Pobject oder Persistent_Object ist gemäß ODMG–93 die Basisklasse aller Datenbankklassen. Die Konstruktoren sind nicht öffentlich, da Pobjekt als abstrakte Klasse verwendet wird. Der Kopierkonstruktor und der Zuweisungsoperator sind privat und führen bei Aufruf zu einem Fehler, da sie mit der Objektidentität nicht vereinbar sind; die versteckte Erzeugung von Objektkopien in C++ könnte sonst zu unerwünschten Effekten führen. Der Aktivierungskonstruktor wurde in Abschnitt H.6.6 erläutert. Die new-Operatoren, sowie odb_activate und odb_deactivate sind im Standard definiert. Zusätzlich übernimmt om_deactivate weitere Aktionen bei der Deaktivie-

rung, die vom Objektmanager benötigt werden. Die destroy-Operation ist notwendig, um Deaktivierung von Objektlöschung zu unterscheiden; sie entfernt das Objekt dauerhaft aus der Datenbank, während der Destruktor nur deaktiviert. _metaClass liefert die Strukturbeschreibung von Pobject im Metaschema und wird in jeder Subklasse überschrieben. Sie wird für Typüberprüfungen und dynamische Zugriffe durch das Anfragesystem gebraucht und verwaltet den Extent, der mit add_to_extent und remove_from_extent aktualisiert wird. Das Datenfeld _selfRef enthält ein Referenzobjekt auf dieses Implementierungsobjekt. Es wird für die Bestimmung der Objektidentität und das Zusammenspiel mit den Referenzklassen benötigt (siehe Abschnitt 4.3). Die generischen Zugriffsfunktionen priv_get_property, priv_set_property, priv_initialize und priv_readout dienen der Interaktion mit dem Anfragesystem und wurden bereits erläutert. Die Funktionen assertCreatePerm, enterMethod und exitMethod überwachen die Benutzerrechte bei Objektzugriffen und dienen der Interaktion mit dem Sicherheitssystem (siehe Abschnitt 4.6.2.5). Das Datenfeld _modified und die zugehörigen Zugriffsfunktionen beschreiben den gegenwärtigen Objektzustand im Vergleich mit der Datenbank: bei unchanged ist das Objekt noch nicht verändert worden, bei locally_changed müssen lokale Objektänderungen noch in die Datenbank geschrieben werden, bei globally_changed sind alle Objektänderungen bereits rausgeschrieben worden und bei invalid ist das Objekt ungültig, da es gerade deaktiviert oder gelöscht wird. Die folgende Klassendefinition zeigt die vollständige Definition der verwendeten Pobjekt-Klasse:

```
class Pobject {
   private:                          Pobject (const Pobject&);
                Pobject&             operator= (const Pobject&);
   protected:   enum                 modify_level { unchanged, globally_changed, locally_changed, invalid };
                modify_level         _modified;                                  // modification state
                static void          assertCreatePerm (InterfaceDef* intf);      // security functions
                void                 enterMethod (const char* meth) const;
                void                 exitMethod () const;
                Ref_Any              _selfRef;
                                     Pobject (InterfaceDef* intf = _metaClass);
                                     Pobject (Activator& act);
                virtual void         priv_destroy ();
                void                 add_to_extent ();
                void                 remove_from_extent ();
                void                 om_deactivate ();
                virtual GFAny        priv_get_property (const char* prop) const;  // generic access functions
                virtual void         priv_set_property (const char* prop, GFAny val);
                virtual void         priv_initialize (GFStruct* values);
                virtual GFStruct*        priv_readout () const;
   public:      virtual void         mark_modified ();
                virtual void         unmark_modified ();
                virtual void         mark_written ();
                virtual bool         is_modified () const;
                virtual bool         is_pending () const;
                static InterfaceDef* _metaClass;                                 // meta information
                virtual InterfaceDef* metaClass () const;
                virtual              ~Pobject ();
                void*                operator new (size_t size, const char* typename = 0);
                void*                operator new (size_t size, Ref_Any cluster, const char* typename = 0);
                void*                operator new (size_t size, const Database* db, const char* typename = 0);
                void                 operator delete (void*);
                virtual void         odb_activate ();
                virtual void         odb_deactivate ();
                virtual void         destroy (); };

typedef class Pobject Persistent_Object;
```

ImportClass

Als gemeinsame Basis für alle Importklassen wird eine Subklasse ImportClass von Pobjekt abgeleitet. Sie aktualisiert die Metainformation (_metaClass) und enthält ein zusätzliches orig-Attribut, das die auf Seite 89 beschriebene globale OID als Rückverweis enthält. Das Attribut und

seine Zugriffsfunktionen sind geschützt, da sie nur intern für die Implementierung von Importklassen und nicht von Anwendungsprogrammen verwendet werden sollen.

```
class ImportClass : public Pobject {
  private:                          ImportClass (const ImportClass&);
                ImportClass&        operator= (const ImportClass&);
  protected:                        ImportClass (InterfaceDef* intf = _metaClass);
                                    ImportClass (Activator& act);
                char*               priv_orig;
                virtual const char* orig () const;
                virtual void        orig (const char* newval);
  public:       static InterfaceDef* _metaClass;
                virtual InterfaceDef* metaClass () const;
                virtual             ~ImportClass (); };
```

DerivedClass

Die Basisklasse für Konvertierungsklassen ist DerivedClass. Neben der aktualisierten Metainformation für sie zwei Funktionen zu Pobject hinzu. Die abstrakte Funktion priv_hasKey überprüft, ob die übergebene Schlüsselinformation mit der eigenen orig-Information übereinstimmt und wird für jede Konvertierungsklasse vom Präprozessor erzeugt. Die Funktion priv_destroy kann jeweils vom Entwickler implementiert werden, wenn die Konvertierungsklasse das Löschen abgeleiteter Objekt unterstützen soll (siehe Abschnitt H.6.7).

```
class DerivedClass : public Pobject {
  private:                          DerivedClass (const DerivedClass&);
                DerivedClass&       operator= (const DerivedClass&);
  protected:                        DerivedClass (InterfaceDef* intf = _metaClass);
                                    DerivedClass (Activator& act);
                virtual bool        priv_hasKey (GFany vkey) const =0; // check if the orig information equals <vkey>
                virtual void        priv_destroy ();                   // class dependent actions for removing the object
                                                                       // persistently from the database (recursion).
  public:       static InterfaceDef* _metaClass;
                virtual InterfaceDef* metaClass () const;
                virtual             ~DerivedClass (); };
```

Programmierschnittstelle

Die Programmierschnittstelle stellt Anwendungsprogrammen einen einfachen Zugriff auf das Sichtsystem zur Verfügung. Die Klassen werden nur kurz beschrieben, auf die zugehörigen Listings wurde verzichtet.

Database, global_DB, local_DB

Database ist die im ODMG-Standard definierte Klasse zum Datenbankzugriff ([Cat94, S.127]). Sie implementiert alle Funktionen zum Öffnen und Schließen einer Datenbank (open, close) und zum Zugriff auf benannte Objekte (set_object_name, get_object_name, rename_object, lookup_object). Im Gegensatz zum Standard sind der Copy-Konstruktor und der Zuweisungsoperator nicht erlaubt, da ihre Semantik unklar ist. Weiterhin werden zusätzliche Zugriffsmethoden angeboten, um die Arbeit mit der Datenbank zu erleichtern (name, status, currentDatabase, currentDBName, get_iss, set_user_pwd).

Die eigentliche Implementierung von Database basiert auf zwei weiteren Klassen, die jeweils die globale und die lokalen Datenbanken darstellen (global_DB, local_DB). Diese sind für alle Netzwerkzugriffe und das Füllen des Metaschemas zuständig und verwalten eine Reihe von Callback-Funktionen für die Interaktion mit den anderen Modulen, zum Beispiel für die Initialisierung des Sicherheitssystems nach dem Öffnen einer Datenbank. Hierzu wird für jede Datenbank ein Objekt des Sicherheitssystems mitgeführt, in dem die Benutzerberechtigungen abgelegt sind, und über das die Anmeldung erfolgt.

Transaction, global_TA, local_TA

Die Klasse Transaction ist ebenfalls in ODMG–93 definiert ([Cat94, S.125]). Über ihre Instanzen werden Transaktionen gestartet und beendet (begin, commit, abort). Die Bedeutung von genesteten Transaktionen und Checkpoints ist im Standard nicht definiert; aus diesem Grunde sind in IRO–DB nur flache Transaktionen erlaubt, wobei eine globale Transaktion aber in jeder externen Datenbank eine entsprechende Subtransaktion laufen hat. Im Unterschied zur Definition in ODMG–93 sind die Kopieroperationen wieder verboten; in Ergänzung zum Standard kann man die aktuelle Transaktion bestimmen (currentTransaction). Die Repräsentation von globalen und lokalen Transaktionen erfolgt über die Klassen global_TA und local_TA. Genau wie die Datenbankklassen verwalten auch die Transaktionsklassen Callback-Listen für die Interaktion mit den anderen Modulen. Mit Hilfe eines Identifikators protokollieren sie ihre Aktivitäten in einem systemübergreifenden Serialisierungsgraphen (siehe Kapitel 4.6.2.4).

IRO

Die Klasse IRO stellt alle wichtigen Funktionen zum Zugriff auf das Sichtsystem als statische Methoden zur Verfügung. Hierzu gehören die Initialisierung und Beendigung des Gesamtsystems (initialize, shutdown), Kommandos zum Öffnen und Schließen von Datenbanken (openDatabase, currentDatabase, closeDatabase), Kommandos zum Starten und Beenden von Transaktionen (beginTransaction, commitTransaction, abortTransaction), sowie eine Funktion zur Anmeldung beim Sicherheitssystem (issLogin).

oql-Funktionen

Zusätzlich definiert ODMG–93 eine Reihe von Funktion zur Auswertung von OQL-Anfragen aus der Sprachbindung heraus. Im Gegensatz zum restriktiven Standard, der die Datentypen der Sprachbindung nicht ausreichend abdeckt, stellt IRO–DB Anfragefunktionen für alle Datentypen zur Verfügung. In [Cat94] sind die folgenden Funktionen definiert:

```
                        int oql (              int& result,   const char* query, ... );
                        int oql (             char& result,   const char* query, ... );
                        int oql (           double& result,   const char* query, ... );
                        int oql (            char*& result,   const char* query, ... );
template<class T>       int oql (        Ref<T>& result,   const char* query, ... );
template<class T>       int oql ( Collection<T>& result,   const char* query, ... );
```

Diese Funktionen entsprechen nicht den in der Sprachbindung verwendeten Datentypen und außerdem gab es bei der verwendeten Compiler-Version Probleme mit den Template-Funktionen, da freie Parameterlisten (*Ellipsen*) in Templates nicht erlaubt sind. Um alle Datentypen abzudekken, sind in unserem System statt dessen die folgenden Funktionen implementiert worden, wobei die Templates durch generische Anfragefunktionen, die das Ergebnis als ungetypte Referenz bzw. im GF-Format zurückliefern, ersetzt worden sind:

```
                        int oql (              short& result,   const char* query, ... );
                        int oql (               long& result,   const char* query, ... );
                        int oql (                int& result,   const char* query, ... );
                        int oql ( unsigned short& result,   const char* query, ... );
                        int oql (  unsigned long& result,   const char* query, ... );
                        int oql (    unsigned int& result,   const char* query, ... );
                        int oql (              float& result,   const char* query, ... );
                        int oql (           double& result,   const char* query, ... );
                        int oql (             char& result,   const char* query, ... );
                        int oql (            char*& result,   const char* query, ... );
                        int oql (  unsigned char& result,   const char* query, ... );
                        int oql (         Ref_Any& result,   const char* query, ... );
                  GFAny oql (      const char* query,   GFAny args );
```

Durch Parametervariablen der Form $rt kann man innerhalb der Anfrage query auf die übergebenen Parameter zugreifen, wobei man die Nummer r des Parameters und seinen Typ t (i=int, c=char,

r=double, s=char*, k=Collection<T>, o=Ref<T>) angeben muß. Der Standard fordert, daß die Typkorrektheit der Parameter zur Laufzeit überprüft wird ([Cat94, S.130]). Das Übergabeverfahren mit der Ellipse läßt es jedoch nicht zu, die Typen der übergebenen Parameter zu bestimmen. Man muß sich darauf verlassen, daß jeder Parameter dem in der Parametervariablen angegebenen Typ entspricht, andernfalls ist das Systemverhalten nicht vorhersagbar. Die korrekte Verwendung des erwarteten Typs innerhalb der Anfrage kann hingegen überprüft werden. Bei der Übergabe muß man beachten, daß C++ Standardkonvertierungen für Ellipsenparameter vornimmt (*integral promotion*, siehe [ElS92] sowie die Unix-Manpage zu stdarg).